Санкт-Петербургский политехнический университет
Петра Великого

А.В. Жуков

Программирование лексического и синтаксического разбора на языках *C*, *Lex* и *Yacc*

Учебное пособие

Санкт-Петербург
2021

Жуков А.В. Программирование лексического и синтаксического разбора на языках C, Lex и Yacc. / учеб. пособие, Изд. 2-е, испр. и доп., 2021. 49 с.

Учебное пособие содержит цикл лабораторных работ по курсу “Транслирующие системы”. Курс предназначен для подготовки бакалавров по направлению 09.03.01 — "Информатика и вычислительная техника". Предмет изучения — программирование лексического и синтаксического разбора на процедурном языке общего назначения, а также с применением стандартных средств описания структуры ввода. В настоящем издании добавлен раздел, касающийся тестирования, а также расширен список заданий.

Стр. 49, табл. 5, библиогр. — 6 назв.

Предисловие

В пособии представлен цикл лабораторных работ по курсу “Транслирующие системы”, в дополнение к циклу лекций [1]. Работы выполняются в ОС Unix, т. к. генераторы трансляторов Lex и Yacc входят в состав ее инструментальных средств,

Таблица 1. Структура пособия

Раздел	Содержание
Введение	На простых примерах продемонстрированы: функции потокового ввода-вывода, трансляция программ на языке C, эффекты от буферизации ввода-вывода и автоматизация тестирования
Темы 1 и 2	Разработка программ лексического и синтаксического разбора на языке C
Тема 3	Разработка программ лексического разбора на языке Lex. Программа состоит из набора правил, каждое из которых содержит шаблон, определяющий класс входных последовательностей (например, букв), и действие на языке C, выполняемое при обнаружении последовательности. Модуль на языке lex обрабатывается одноименным транслятором, а результат (модуль на языке C) — командой **cc**
Тема 4	Разработка программ синтаксического разбора на языке Yacc, совместно с модулем лексического разбора на языке Lex. Программа составляется в виде набора правил, в общем случае рекурсивных, которые определяют нетерминальные символы (конструкции языка) через терминальные символы (лексические единицы) и ранее определенные нетерминальные символы. В правилах могут быть заданы действия на языке C. Модуль на языке lex вызывается из yacc-программы для чтения лексем из входного потока
Приложение 1	Варианты заданий
Приложение 2	Служебные литеры в регулярных выражениях Lex
Приложение 3	Контрольные вопросы по Lex
Приложение 4	Десятичные коды ASCII — для отладки программ на Lex/Yacc
Приложение 5	Особенности работы в DOS/WinXP
Послесловие	Краткая характеристика дополнительных примеров в прилагаемом архиве и перечень наиболее существенных вопросов, не включенных в пособие

Файлы с примерами программ находятся в каталоге works. Все, что написано на языке C, то есть примеры из введения и тем 1 и 2, находится в каталоге works/c; примеры к теме 3 — в works/lex, к теме 4 — в works/yacc.

В отчеты по всем темам входят:

- тесты для всех примеров ввода-вывода, включенных в задание;
- индивидуальное задание: его номер, формулировка, исходный текст программы с пояснениями, синтаксические диаграммы.

Введение

Программы лексического и синтаксического разбора, написанные с использованием языков lex и yacc, выполняют ввод-вывод через стандартные потоки. По умолчанию входной и выходной потоки связаны с консолью оператора (ввод с клавиатуры, вывод на дисплей), но могут быть перенаправлены:

./a.out <text.in >text.out

При таком вызове программа a.out через стандартный входной поток читает данные из файла text.in; а то, что она выведет в стандартный выходной поток, запишется в text.out. Перенаправление реализовано на уровне операционной системы; от программы требуется только, чтобы для ввода-вывода использовались функции, определенные в stdio.h. Именно так написаны примеры лексического и синтаксического разбора на языке C (темы 1 и 2).

Рассмотрим примеры из каталога c/intro, которые демонстрируют свойства ввода-вывода с использованием потоков.

Буферизация ввода-вывода

Вывод в стандартный выходной поток буферизуется построчно. Проверим это на простом примере.

Листинг 1.1 (buf_out.c). Буферизация вывода

```
#include <stdio.h>

main ()
{
    int i;


    for (i = 0; i < 5; i++) {
        printf("i = %d\t", i);
    }
    sleep(2);
    putchar('\n');
    return 0;
}
```

Выполните трансляцию файла: **make buf_out**. Обнаружив в текущем каталоге файл buf_out.c, команда **make** вызовет стандартную утилиту компиляции **cc**. В результате будет получен исполняемый модуль buf_out. Вызовите его: **./buf_out**.

Отображение на экране появилось после двух секунд паузы. Пока выполнялся sleep, на экране было пусто, потому что вывод printf остался в буфере вывода. Результат printf будет отправлен из буфера на консоль по одному из событий:

1. вывод '\n' (символ конца строки);
2. вызов fflush(stdout);
3. безаварийное завершение программы;
4. переполнение буфера.

Завершите программу нажатием <**Ctrl+C**>; экран вывода должен быть пустым, т. к. это завершение — аварийное. Проверьте вариант 3: вновь вызвав программу, завершите ее нажатием <**Enter**>. Проверьте вариант 2: вставьте вызов fflush(stdout) либо в цикл после printf, либо после цикла перед sleep. Наконец, проверьте первый вариант: уберите вызовы fflush и замените табуляцию '\t' литерой конца строки.

Следующий пример демонстрирует буферизацию ввода.

Листинг 1.2 (buf_in.c). Буферизация ввода

```
main ()
{
    int c;


    do {
        c = getchar();
        printf("<%c (%d)>\n", c, c);
    } while (c != EOF);
    printf("\n---< Normal shutdown >---\n");
    return 0;
}
```

Получите исполняемый модуль buf_in, вызовите его и введите любые цифры или буквы. На экране видна только эхо-печать, а вывод, заданный printf, отсутствует. Если бы функция printf, стоящая после getchar, была вызвана, мы бы увидели ее результат на экране (т. к. выводимая строка заканчивается '\n'). Значит, мы задержались на вызове getchar.

Причина в том, что ввод тоже буферизуется, и функция getchar не вернет управление, пока не получит '\n' (<**Enter**>), после чего она будет возвращать литеры из буфера ввода (не ожидая нажатий клавиш) до полной очистки буфера.

Выход из цикла произойдет, когда getchar вернет код завершения потока EOF. Этот код вводится клавишей <**Ctrl+D**>, но только если буфер ввода *пуст*. Если в буфере что-то есть, нажатие <**Ctrl+D**> равносильно вызову fflush(stdout); код EOF при этом теряется. Итак, нажатие <**Ctrl+D**> выводит строку из буфера или (если буфер пуст) генерирует код EOF. Проверьте реакцию buf_in на нажатия клавиш <**Enter**> и <**Ctrl+D**>; это пригодится вам при тестировании примеров.

Перенаправление

Стандартные потоки можно перенаправить: то есть связать их с файлами, отключив от консоли. Перенаправление задается, без каких-либо изменений в программе, в командной строке при помощи символов < и >. Символ > означает выходной поток, в программах на С он обозначается как stdout. Символ < — поток ввода, обозначается stdin.

Пример, с использованием результата трансляции программы buf_in.c:

./buf_in <test.in >test.out

Еще один поток, часто применяемый в Unix для вывода дополнительных сведений: предупреждений, сообщений об ошибках или справочной информации — это стандартный поток stderr. В командной строке он обозначается 2>. Для вывода в stderr из программы на языке С используется функция fprintf.

Листинг 1.3 (buf_in2.c). Вывод сообщений в stderr

```
main ()
{
    int c;

    fprintf(stderr, "Enter string: "); fflush(stderr);
    do {
        c = getchar(); printf("<%c (%d)>\n", c, c);
    } while (c != '\n' && c != EOF);
    fprintf(stderr, "\n---< Normal shutdown >---\n");
    return 0;
}
```

Выполните трансляцию buf_in2.c и проверьте вызовы:

./buf_in2 <test.in
./buf_in2 <test.in >test.out
./buf_in2 <test.in >test.out 2>test.err
./buf_in2 <test.in 2&>test.out

Третий вариант вызова, с раздельным перенаправлением потоков stdout и stderr, удобен при тестировании примеров на тему lex и yacc, когда включен *отладочный* режим. Вывод этих программ по умолчанию идет в stdout, а отладочные сообщения направляются в stderr. Можно воспользоваться и вторым вариантом — тогда вывод пойдет в файл, а трасса отобразится на консоли.

Оператор 2&> объединяет выходные потоки stdout и stderr, направляя их в один файл.

В первом и четвертом варианте потоки stdout и stderr смешиваются: оба идут или на консоль, или в файл. Для трассировки это неподходящий вариант, но он может пригодиться при получении сведений о параметрах вызова программ (например, стандартных утилит Unix), поскольку неизвестно заранее, как там запрограммирован вывод — через stderr или через stdout.

Тестирование

Чтобы проверить программу для обработки текста, надо подать на вход текстовые данные и убедиться, что вывод правильный. После каждого изменения в программе надо заново выполнить все предыдущие тесты, т. к. при исправлениях могли внести ошибку, которой раньше не было[1]. Но каждый раз запускать тесты вручную, а вывод проверять визуально — это и утомительно, и ненадежно; тут нужна автоматизация.

В нашем случае, когда ввод текстовый и вывод текстовый, автоматизация проста (см. каталог test_automation). Входные данные записываем в файлы с расширением **in**. Правильные выходные данные сохраняем в одноименных out-файлах. В этом же каталоге находится сценарий **x.sh**, который для всех пар одноименных in- и out-файлов вызовет проверяемую программу **test**, связав входной поток с in-файлом, а вывод направит в файл с расширением **result**. Затем оценивается разница в данных между **result-** и **out**-файлами и, в случае расхождения, выдается сообщение. Независимо от исхода очередного теста сценарий будет выполнен для всех пар in- и out-файлов.

Листинг 1.4 (x.sh). Тестовый сценарий

```
#bash

prg="./test"
ext="result"  # extension of result file

rm -f *.$ext                    # clean up previous results
for f_in in $(ls -t *.in)       # time sorted list of in-files
do
  f_out=$(basename $f_in in)"out"          # out-file name
  if [ -f $f_out ] ;                       # if exists, run test
  then
        tmp=$f_out"."$ext                  # result file name
        echo $f_in "->" $tmp "=?" $f_out   # show the cmd
        $prg <$f_in 2&>$tmp                # run the cmd
        diff -q $tmp $f_out                # compare files
  fi
done
```

[1] Regression test — это набор тестов, пополняемый и повторяемый при каждом изменении программы.

Обратите внимание: при вызове **test** потоки stdout и stderr объединены. Это сделано для того, чтобы не пришлось для каждого in-файла хранить пару выходных: **-out** и **-err**.

Входные файлы обрабатываются в историческом порядке (их список отсортирован по дате создания). Программа **test** вызывается лишь тогда, когда для in-файла существует out-файл с тем же именем. Перед каждым вызовом **test** на консоли отображаются имена файлов, участвующих в обработке. Ключ –q в команде diff (сравнение пары файлов) означает, что при совпадении данных ничего не выводится.

Продемонстрируем применение **x.sh** на примере программы **test.c**.

Листинг 1.5 (test.c). Копирование текста до точки

```
#define sz 1024
char buf[sz];

main()
{
    int bcnt;
    char *p;


    bcnt = fread(buf, 1, sz-1, stdin);
    buf[bcnt+1] = 0;
    p = strchr(buf, '.');      // p = NULL if not found
    if (p) {
      bcnt = p - buf;
    }
    fwrite(buf, 1, bcnt, stdout);
    return 0;
}
```

Если на входе задан текст, не содержащий точки, то он копируется на выход целиком.

Примечание:

Параметры библиотечной функции fread: buf — массив для сохранения результата чтения, 1 — размер элемента данных в байтах, sz-1 — число элементов, stdin — поток для чтения. Результат fread — число прочитанных элементов (или отрицательный код ошибки). Это число может быть меньше sz-1, если при чтении встретился конец потока.

После компиляции программы **test.c** запустите **x.sh**, предварительно пометив его как исполняемый[2]: **chmod 777 x.sh**.

Замечание: Возможно, что **x.sh** не будет отработан из-за сбоя на операторе **do**. Причина в том, что **x.sh** правили в редакторе для Windows. Конец строки в Unix обозначается кодом 0xa, а в DOS/Windows перед 0xa ставится еще 0xd. Чтобы избавиться от лишних кодов 0xd, выполните **dos2unix x.sh**.

При запуске **x.sh** должны быть отработаны четыре теста, по числу пар in- и out-файлов. Для пятого in-файла нет пары, создайте ее сами и заново выполните **x.sh**.

Затем добавьте **x6.test.in** и **x7.test.in** размером 1023 и 1024 байт, взяв за основу копию какого-нибудь исходного теста достаточной длины. Поскольку функция fread в **test.c** читает максимум 1023 байт, текст за пределами этой границы не анализируется и на выход не идет. Причем программа не сообщает об этом!

[2] Текстовые файлы, созданные в Windows, после импорта в Linux выглядят как исполняемые (помечены звездочкой); исправить это недоразумение можно командой **chmod 666 file**.

Добавьте в **test.c** проверку: если размер входного потока больше 1023 байт, в stderr выводится сообщение "?-input file size > 1023", а в stdout — ничего. Создайте **x6.test.out** и **x7.test.out** и проверьте весь тестовый набор заново.

Теперь удалите исполняемый файл **test** и снова вызовите **x.sh**. Результат странный, т. к. цикл отработан в отсутствие тестируемой программы. Добавьте в **x.sh**, до входа в цикл, проверку наличия файла $prg, по аналогии с проверкой существования out-файла.

В последующих работах тестирование должно быть организовано по рассмотренной схеме, с участием сценария **x.sh**.

В завершение вернемся к Листингу 1.2 и проверим, как C обрабатывает парные коды 0xd и 0xa в конце строк — для файлов, созданных в DOS/Windows.

Выполните команду **uinx2dos test.in**, и убедитесь, что в **test.in** появились коды 0xd (для просмотра hex-кодов есть редактор **hexedit**, а в **Midnight Commander** — **F3** (view), а затем **F4** (hex)). Затем вызовите **./buf_in2 <test.in** и проверьте, выведены ли коды 0xd на консоль. Если да, то результаты тестирования текстов, внешне идентичных, но созданных в DOS/Windows и в Unix, будут отличаться. А поскольку код 0xd невидим, причина сбоя будет непонятной.

Примечание: для массового преобразования файлов в дереве каталогов, начиная от текущего, может пригодиться сценарий **tree.sh**.

И в завершение еще раз вернемся к тесту **x7.test** и вспомним, для чего предназначена программа **test**. Что если размер файла > 1023, но среди этих первых 1023 байт есть литера точки?

Создайте **x8.test** для этой ситуации и скорректируйте программу **test**.

Тема 1. Программирование лексического разбора на языке *C*

Образец программы лексического разбора приведен в модуле scanner. Он состоит из двух файлов: scanner.c и scanner.h. Модуль test_scanner.c содержит функцию main, в ней запрограммирован циклический вызов функции yylex (она реализована в модуле scanner). Базовый вариант модуля scanner, вместе с тестовой программой, находится в папке c/scanner.

Трансляция — командой **cc *.c**, если в текущем каталоге нет никаких других модулей на C. Ввод и вывод идут через стандартные потоки (см. Введение).

Функции в составе модуля *scanner*

В состав модуля scanner входит глобальная функция для лексического разбора.

- int ***yylex*** (void) — читает литеры из входного потока и, пропуская вначале разделители, выявляет числа (код лексемы — NUM) и идентификаторы (код лексемы — ID); функция возвращает одиночные литеры, не являющиеся буквами или числами, как *литералы* (код лексемы равен ASCII-коду литеры); лексема с кодом 0 означает конец входного потока.
- int ***prn_token*** (int) — выводит в stderr код указанной лексемы; видимые литералы печатаются как литеры, а для невидимых литералов выводится их десятичный код.

Примечания:

- Функция yylex построена на основе программного цикла с ветвлением по значению переменной состояния.
- Лексический разбор выполняется, в основном, в локальной функции __yylex, которая вызывается из yylex. Исправления при выполнении задания следует делать в __yylex.

Глобальные данные модуля *scanner*

Модуль scanner предоставляет следующие глобальные данные:

- ***yytext*** — массив литер, в котором формируется (накапливается) текст очередной лексемы при работе функции yylex; строка в yytext завершается нулем, т. о. yytext можно выводить функцией printf и обрабатывать строковыми функциями из библиотеки C.
- ***yyleng*** — длина строки, сформированной в yytext.
- ***yylval*** — семантическое (сопутствующее) значение лексемы, формируется в yylex при получении числа.

Эти переменные обозначены в заголовочном файле scanner.h. Там же определены коды лексем NUM и ID. Коды лексем должны быть за пределами диапазона литералов [1..255]. Ноль также зарезервирован — для признака конца ввода.

Примечание:

Названия глобальных объектов модуля scanner и определения этих объектов следуют соглашениям, принятым для стандартной утилиты lex.

Реализация функции разбора *__yylex*

Функция работает в бесконечном цикле, начиная с состояния state = 0. На каждой итерации считываем литеру из входного потока, заменяя признак EOF (-1) нулем. (Литера определена не как char, а как int, поскольку EOF — это -1 в 32-битном представлении.)

В исходном состоянии (state = 0) пропускаем литеру, если это разделитель. Как только в состоянии 0 встретится что-то другое, выясняем, *началом* чего оно является. Если получена буква, то это начало идентификатора, и переходим в состояние 1. Если цифра, то это начало числа, и переходим в состояние 2. Если не буква и не цифра, то считаем это литералом (частный случай — 0, признак конца ввода) и сразу возвращаем ASCII-код, записав его также в yytext.

В состояниях 1 или 2 остаемся, накапливая литеры в yytext, пока не получим что-либо не относящееся соответственно к идентификатору или числу. Эту литеру возвращаем во входной поток и выходим из функции с кодом лексемы NUM или ID. Если накапливали число, то перед выходом вызываем функцию convert для преобразования строки yytext в число yylval, с проверкой переполнения.

Примечание: проверка переполнения в convert двукратная: первая (errno == ERANGE) обнаруживает нарушение диапазона типа long long (наибольшего целого, поддерживаемого системой), а вторая (yylval != x) — что, хотя результат и в диапазоне long long, он все же превышает пределы переменной yylval, которая может быть меньшей размерности.

Главный модуль *test_scaner*

Здесь запрограммирован циклический вызов yylex до получения признака конца ввода, а также вызов функции prn_token, которая выводит в stderr код указанной лексемы; видимые литералы печатаются в символьном формате, а невидимые — в формате десятичного числа.

Примеры модернизации модуля *scanner*

В signed_num.txt записан вариант функции __yylex для распознавания целых чисел *со знаком*. Теперь литера '-' с примыкающими к ней цифрами воспринимается как одно целое. Отдельно стоящая литера '-' по-прежнему распознается как литерал, а последовательность цифр без предшествующего знака '-' — как лексема NUM.

Проверьте этот вариант; убедитесь, в частности, что контроль переполнения для отрицательных чисел работает.

Рассмотренный пример очень простой, т. к. решение о том, отнести ли знак '-' к числу или счесть его литералом, принимается в начале разбора, и оно окончательное. Сложнее разбирать числа в разных системах счисления и/или в разной нотации, т. к. по ходу разбора приходится уточнять первоначальное решение или даже пересматривать его, *возвращая* прочитанное во входной поток (откат).

Предположим, нужно распознавать, наряду (!) с десятичными числами, *двоичные* числа в нотации ассемблера a86. Пример: 0101xb.

Можно попытаться решить эту задачу, не вводя дополнительных состояний: усложнить логику __yylex в состоянии state = 1 (разбор *десятичных* чисел). Ниже рассмотрено решение в этом стиле (без расширения множества состояний), и оно неправильное.

По первой литере (0 или 1) выбрав state = 1, читаем до литеры, отличной от 0–9. Если это не 'x', принятую последовательность считаем десятичным числом, а последнюю литеру возвращаем на вход — все как обычно. Если же это 'x', считываем еще одну литеру. Если это 'b', то цифры в накопленной последовательности считаем двоичным числом.

Если после 'x' не 'b', принятая последовательность не является записью двоичного числа. Скорее уж десятичного — до буквы 'x' (тогда 'x' — начало следующей лексемы). Делаем откат на две литеры ('x' и 'b'). Оставшаяся часть yytext — десятичное число, а все, что за ним, будет распознано при следующем вызове __yylex. Например, ввод 0101x= распадется на десятичное число 0101, идентификатор x и знак равенства.

В чем ошибка? Представьте себе, как будет воспринят ввод 0102. При получении цифры 2–9 после нулей и единиц придется изменить решение: это число не двоичное, а десятичное.

Решите эту задачу, введя два дополнительных состояния: одно для чтения *двоичного* числа[3], до литеры 'x', и еще одно для чтения завершающей литеры 'b'. Это — первое задание по теме 1.

Второе, индивидуальное задание по теме 1 см. в *Приложении 1*.

[3] В это состояние попадаем, если первая цифра двоичная, и остаемся в нем, пока приходят двоичные цифры.

Тема 2. Программирование синтаксического разбора на языке *C*

В цикле синтаксического разбора мы вызываем функцию yylex и принимаем решение исходя из полученного кода лексемы. Лексема с кодом 0 означает конец входного потока. Как правило, решение (даже в случае завершения входного потока) зависит от предыстории. Одна и та же лексема, в зависимости от контекста, может быть допустима или нет — и тогда разбор должен прекратиться с выводом ошибки. Например, в списке чисел, разделенных запятыми, ошибка — это подряд два числа или две запятые, или конец ввода после запятой.

Пример программы синтаксического разбора приведен в каталоге c/parser. Сначала рассмотрим вспомогательные функции в составе модуля parser, предназначенные для проверки лексемы и аварийного завершения программы.

Функции в составе модуля *parser*

Модуль parser предоставляет следующие глобальные функции для программирования синтаксического разбора:

- int ***chk_token*** (int, int *) — с помощью локальной функции in проверяет, входит ли число (код лексемы) в массив целых чисел (лексем), ограниченный нулем; если проверка успешна, возвращает входной код лексемы, а иначе отображает сообщение об ошибке и код лексемы (с помощью функции prn_token) и завершает программу.
- int ***rd_token*** (int *) — вызывает функцию yylex, а затем проверяет, входит ли полученный код лексемы в массив допустимых лексем, указанный в аргументе; для проверки использует функцию chk_token; чтение недопустимой лексемы приводит к аварийному завершению.
- void ***bad_eof*** (void) — выводит сообщение о недопустимом завершении входного потока и прекращает программу.

Пример синтаксического анализатора *parse_0*

Структура ввода в примере — список чисел, разделенных запятой или точкой с запятой. Пустой список и список из одного элемента допускаются. Результат разбора — вывод числа элементов и среднего значения.

Листинг 2.1 (parse_0.c). Разбор списка чисел

```
int chk_1[] = { NUM, 0 };
int chk_2[] = { ',', ';', 0 };

int main (void)
{
    int token, counter, total;

    if (!rd_token(chk_1)) {          /* get number or EOF */
        printf("Empty list\n");      /* EOF is OK here */
        return 0;
    }
    for (counter = total = 0;;) {
        counter++; total += yylval;
        if (!rd_token(chk_2))        /* get comma (or EOF) */
            break;                   /* end of list, EOF is OK */
        if (!rd_token(chk_1))        /* get a number (or EOF) */
            bad_eof();               /* EOF not allowed here! */
    }
    printf("no. of items = %d, average = %d", counter, total/counter);
    return 0;
}
```

Замечание:

Для компиляции этого примера, в папке c/parser должны быть также scanner.c и scanner.h из папки c/scanner. Не надо их копировать — достаточно создать в c/parser символические ссылки на них[4]. В Midnight Commander для этого надо встать на scanner.c и выбрать в меню **File** команду **SymLink**, затем повторить для scanner.h. Или вручную: командой **ln –s**.

Синтаксический разбор запрограммирован в функции main. Подсчет элементов списка ведется в переменной counter, а сумма, необходимая для итогового вычисления среднего, накапливается в total.

Сначала считываем первую лексему, ожидая число или конец ввода. Эти ожидания определены в массиве chk_1, который указан в первом вызове rd_token. Все прочие лексемы считаем ошибкой.

Замечание:

Код 0 (конец ввода) всегда указывается в конце последовательности допустимых лексем (см. chk_1, chk_2) — он ее ограничивает, и он же является признаком конца ввода. Поэтому для функции rd_token конец ввода не ошибка. Когда rd_token возвращает 0, разбор закончен. Программа завершается либо оператором return, если EOF не нарушает структуры ввода, либо, в противном случае, вызовом bad_eof.

Если при первом вызове rd_token в примере мы получили лексему 0, то это не ошибка, т. к. по условию задачи *пустой список* возможен. Разбор завершится сообщением о том, что список пуст. Если же первый вызов rd_token вернул не ноль, значит, получена лексема NUM — без вариантов, потому что во множестве chk_1 больше ничего нет. (Любая другая лексема не прошла бы проверку в rd_token, и программа завершилась бы аварийно.)

Затем следует бесконечный цикл чтения *непустого* списка. В начале каждой итерации у нас есть число и его значение в yylval. Прибавив yylval к total и увеличив счетчик элементов, читаем следующую лексему. Ожидается знак препинания (и, как всегда, признак конца ввода), но не число! Если получим 0 (конец ввода), то разбор закончен успешно.

Если же получен знак препинания, продолжаем чтение, ожидая только число. Список не должен заканчиваться знаком препинания, поэтому при получении признака EOF вызываем функцию bad_eof.

Замечание:

В этом примере переменная token нам не понадобилась, поскольку структура ввода однозначна: за числом следует запятая, за запятой — число. В более сложных случаях (например, если список разделен знаками арифметических операций и требуется вычислить заданное таким образом выражение), мы сначала присваиваем token = rd_token(...), как всегда проверяем на 0 (конец ввода), а затем принимаем решение в зависимости от кода полученной лексемы.

[4] При удалении символической ссылки файл-источник остается; напротив, удалив все жесткие ссылки на файл, вы в конце концов удалите и сам файл.

Тема 3. Программирование лексического разбора на языке *lex*

Lex — это генератор программ лексической обработки текстов. Основу исходной программы на языке lex составляет таблица регулярных выражений, или *шаблонов*, и соответствующих им *действий*, которые задаются пользователем в виде фрагментов на языке C.

Исходная программа транслируется посредством утилиты lex в модуль на языке C, в котором определена глобальная функция yylex. Каждое обращение к yylex возобновляет обработку текущего входного потока до получения очередной лексемы; при обнаружении лексемы yylex выполняет действие, связанное с шаблоном, который распознал лексему. Цикл обращений к yylex программируется отдельно; он должен завершаться при возвращении yylex нулевого результата (конец входного потока).

Если функция yylex не смогла поставить в соответствие текущему входному потоку ни один из шаблонов, выполняется действие по умолчанию: очередная литера копируется в выходной поток.

Рассмотрим программу, которая передает в выходной поток все литеры входного потока кроме пробелов и/или табуляций в начале строки.

Листинг 3.1 (ex1.l). Удаление пробелов и табуляций в начале строк

```
%%
^[ \t]+    ;
%%

#ifndef yywrap
int yywrap() { return 1; }
#endif

main () { while (yylex()); }
```

Между разделителями "%%" заданы *правила*, каждое в форме шаблон-действие. Единственное правило содержит шаблон "^[\t]+" и пустое действие ";". Квадратные скобки задают одну литеру из набора литер в скобках, т. е. "[\t]" означает пробел *или* табуляцию. Служебная литера "+" задает ненулевое число повторений, а "^" — начало строки.

Поскольку действие пустое, то последовательности, соответствующие этому шаблону, игнорируются. Литеры, не распознанные ни одним правилом, передаются в выходной поток.

Выполните трансляцию командой **make ex1**. После вызова **./ex1** введите две строки: без начальных пробелов и с пробелами. Ввод и вывод выполняются через стандартные потоки, и можно использовать перенаправление: **./ex1 <test.in >test.out**.

Примечания:

- Текст после второго разделителя "%%" при трансляции переписывается без изменений в конец C-программы, сгенерированной lex. Здесь обычно задают функции, в том числе main и yywrap, которые определяют точку входа в программу и реакцию программы на завершение входного потока.
- В последующих листингах определение yywrap опущено, т. к. оно везде одинаково. Функция main показана в тех случаях, когда в нее добавлены некие предварительные и/или итоговые действия. По умолчанию считаем, что yywrap и main определены так, как в файле yy.c, который используется в большинстве примеров.

Далее приведено описание языка lex. Предварительно дан краткий обзор справочного характера; в дальнейшем новые понятия рассматриваются более подробно, с примерами.

Структура и синтаксис программы на языке *lex*

Общая форма исходного текста lex-программы:

определения
%%
правила
%%
процедуры пользователя

Обязательна только секция правил; она ограничивается парой разделителей "%%" даже при отсутствии других секций.

Секция определений

Секция определений может содержать в любой последовательности:

* макроопределения регулярных выражений, без отступов:

```
{name}  pattern
```

* включаемый код на языке C, с отступом:

```
code
```

* включаемый код на языке C, без отступов[5]:

```
%{
code
%}
```

* стартовые условия, без отступов:

```
%S cond1, cond2 ...
```

* комментарии в стиле языка C.

Листинг 3.2. Пример секции определений

```
{digit} [0-9]
    int count = 0;
%{
#include <stdlib.h>
#define YY_USER_ACTION trace();
void skip_comments();
%}
%S quotes, newPage
/* macro, code, code, start conditions, comment */
```

Секция правил

Правила задаются без отступа, каждое в форме "шаблон действие".

Действие — это один оператор языка C; здесь допускается *составной* оператор, т. е. последовательность операторов через точку с запятой, заключенная в фигурные скобки, или даже последовательность операторов через запятую[6]. В любом случае действие может быть записано на нескольких строках.

В шаблонах могут использоваться обычные и служебные литеры (*Приложение 2*).

В начале секции правил можно задать, с отступом, фрагмент на языке C. Выглядит это как правило без шаблона — только действие. Этот фрагмент при трансляции копируется в инициализирующую часть C-программы и будет выполнен один раз ее запуске.

[5] Для функций, структура которых отлична от int func(void), в этой секции должны быть заданы их прототипы (см., например, skip_comments в листинге 3.2).

[6] Конструкция x = 1, y = 2; в языке C считается разновидностью *простого* оператора. Напротив, конструкция { x = 1; y = 2; } — *сложный* оператор. В любом случае это один оператор, что и требуется для программирования действия в lex.

Секция процедур

Все, что идет за вторым разделителем "%%", передается в C-код без изменений. Обычно здесь задают пользовательские функции, такие как:

- main — точка входа в C-программу;
- yywrap — вызывается при завершении входного потока; если она вернет единицу, то разбор закончится.

Правила

В этом разделе рассмотрены регулярные выражения, действия и управление правилами.

Регулярные выражения

Шаблоны, определяющие классы искомых последовательностей литер, записываются с применением регулярных выражений. (Термины "шаблон" и "регулярное выражение" в дальнейшем используются как синонимы.)

В языке lex принята следующая нотация:

* Последовательность литер, не содержащая служебных операторов, задает себя буквально. Например, шаблон для сопоставления со словом “integer”:

```
integer
```

* Для включения пробельных литер в шаблон всю последовательность надо заключить в двойные кавычки. Так, последовательность “silly thing” может быть задана шаблоном:

```
"silly thing"
```

* Оператор '*' означает ноль или более повторений. Например, пустая последовательность и последовательность литер 'm' могут быть заданы одним шаблоном:

```
m*
```

* Оператор '+' означает одно и более повторений. Например, непустая последовательность литер 'm' задается выражением:

```
m+
```

* Выражению, за которым следует '?', соответствует 0 или 1 экземпляр этого выражения (т. е. выражение необязательно). Например, необязательное 'a' перед 'b', можно задать как:

```
a?b
```

* Точка соответствует любой литере кроме новой строки. Например, последовательность из пяти литер, которая начинается с 'm' и заканчивается 'y', может быть обозначена как:

```
m...y
```

* Альтернатива обозначается '|'. Например, совпадение с 'love' или с 'money' можно задать так:

```
love|money
```

* Выражения могут быть сгруппированы с использованием скобок '(' ')'. Например, последовательность двоичных цифр, за которой следует литера 'b', может быть задана как:

```
(0|1)+b
```

* Знак '^' перед шаблоном означает, что шаблон должен быть выявлен в начале строки. Следующее правило соответствует слову 'Word' в начале строки:

```
^Word
```

* Знак '$' в конце шаблона задает сопоставление в конце строки. Следующее правило соответствует слову 'times' в конце строки:

```
times$
```

* Чтобы шаблон был распознан какое-то число раз подряд, это число нужно указать после шаблона в фигурных скобках. Так, чтобы выявить 'qupuqu', можно использовать:

```
(qu){3}
```

* Чтобы задать число повторений в некотором диапазоне, после выражения записываются два числа в фигурных скобках, через запятую. Так, чтобы выявить 3, 4 или 5 повторений 'ho', т. е. 'hohoho', ' hohohoho' или ' hohohohoho', используйте:

```
(ho){3,5}
```

* Если число повторений должно быть не меньше некоторого предела, то в фигурных скобках после выражения записывается одно число с запятой. Так, чтобы выявить не менее двух повторений 'oops', задайте:

```
(oops){2,}
```

* Набор литер в квадратных скобках '[' ']' означает любую из этих литер. Например, чтобы задать произвольную литеру из множества ' ', '\t' и '\n', используйте:

```
[ \t\n]
```

Внутри квадратных скобок только три литеры являются служебными: '\', '-' и '^'.

- Литера '^' в самом начале задает любую литеру *не* из этого множества. Например, для задания чего угодно кроме 'a', 'b' и 'c' используйте:

```
[^abc]
```

- Диапазоны задаются через дефис. Например, любая цифра или буква, прописная или строчная может быть задана так:

```
[0-9A-Za-z]
```

* Регулярные выражения могут объединяться. Например, следующее выражение выявляет идентификатор (начинается с буквы, за которой следует ноль или более букв и/или цифр):

```
[a-zA-Z][0-9a-zA-Z]*
```

* Чтобы служебные литеры воспринимались буквально, их заключают в двойные кавычки или ставят перед каждой знак '\'. Любое из выражений ниже может быть использовано для сопоставления с литерой '*', за которой следует одна или более цифр:

```
\*[0-9]+
"*"[0-9]+
```

- Буквальное задание литеры '\' возможно в двух вариантах:

```
\\
"\"
```

- Для задания новой строки, табуляции и т. п. используются обозначения, принятые в языке C:

`\n` — конец строки
`\t` — табуляция

* Литера '/' задает правый, или "концевой" (trailing) контекст: выявляется то, что задано слева от '/', но только если к нему примыкает заданное справа от '/'. Например, '4', если за ней следует 'you', можно определить так:

```
4/you
```

Действия

Действие — это оператор языка C, выполняемый при успешном сопоставлении ввода с шаблоном.

Пустое действие и действие по умолчанию

Простейшее действие — это *пустое* действие, которое по правилам языка C задается в виде ';'. Входной текст игнорируется, т. е. не идет на выход и не сохраняется в данных.

Литеры, не соответствующие ни одному шаблону, передаются на выход — это действие *по умолчанию*. Например, часто используемое правило (см, ниже) не пропускает на выход

пробельные литеры (пробел, табуляцию и новую строку). Если задано только это правило, то все другие литеры передаются с входа на выход,

```
[ \t\n]   ;
```

В следующем примере распознаются *все* литеры, так что действие по умолчанию нигде не используется. Вывод результата выполняет функция main после окончания циклов вызова yylex, т. е. при завершении входного потока.

Листинг 3.3 (ex2.l). Подсчет числа строк

```
    int lineno = 0;
%%
\n    lineno++;
.     ;
%%

main()
{
    while( yylex() );
    printf( "%d lines\n", lineno );
}
```

Если необходимо, чтобы программа не пропускала на выход непонятные ей литеры, то действие по умолчанию нужно блокировать, указав ключ **-s** при вызове lex. Проверим это на примере ex1.l, где действие по умолчанию превалирует. Выполните **lex -s ex1.l** и **cc ex1.c**. Скорей всего вы получите предупреждение уже при трансляции; выполнение ./ex1 с файлом test.in закончится сообщением “flex scanner jammed” (сканер заклинило).

Обратите внимание, что в примере ex2.l есть правила и для '\n', и для точки — она означает любой символ *кроме* \n. То есть программа распознает *все* литеры, и поэтому нет нужды задавать ключ **-s**[7].

Доступ к элементам входной последовательности

Распознанная входная последовательность литер сохраняется в массиве yytext; ее длина записывается в переменную yyleng.

Пользователь может исправлять содержимое yytext в пределах первых yyleng позиций. Первая литера найденной строки доступна как yytext[0], а последняя — как yytext[yyleng-1].

В следующем примере задан подсчет последовательностей, которые обозначают знаковые целые числа; каждый раз при обнаружении такой последовательности выводится текущее значение счетчика чисел и текст лексемы.

Листинг 3.4 (ex3.l). Подсчет и вывод знаковых целых чисел

```
%{
int count = 0;
%}

%%
[-+]?[0-9]+  {
                count++;
                printf( "%d %s\n", count, yytext );
             }
%%
```

[7] Ключ -s не рекомендуется стандартом POSIX.

Вывод yytext — это настолько частое действие, что для него определена макрокоманда ECHO. В следующем примере в выходной поток передаются идентификаторы и беззнаковые числа, по одному на строке, а все прочее отсеивается. Литера '|' справа от шаблона означает "то же действие, что и для следующего правила".

Листинг 3.5 (ex4.l). Вывод идентификаторов и беззнаковых целых чисел

```
%%
[0-9]+      |
[a-zA-Z]+   { ECHO; printf( "\n" ); }
.|\n        ;
%%
```

В этом примере уже достаточно много правил, чтобы проверить отладочный режим lex. Выполните трансляцию: **lex –d ex4.l** и **cc ex4.c** — и протестируйте программу ex4.

Использование переменной yyleng показано в программе подсчета идентификаторов по длине. Результат — гистограмма длин слов в диапазоне от 1 до 40, в виде текста. Обратите внимание на *первое* правило, которое состоит только из действия. Это действие выполняется один раз при запуске программы.

Листинг 3.6 (ex5.l). Подсчет и вывод гистограммы длин слов

```
    int len[40], i;
%%
        {
          for( i = 0; i < 40; i++ )
            len[i] = 0;
        }
[a-z]+  len[yyleng]++;
.|\n    ;
%%

main()
{
    while( yylex() );
    for( i = 0; i < 40; i++ )
        if( len[i] > 0 )
            printf( "%5d%10d\n", i, len[i] );
}
```

Функции *yymore* и *yyless*

Функции yymore, yyless(n) дают дополнительные возможности по управлению yytext:

- yymore — отключает режим перезаписи для следующего (одного) сопоставления, т. е. литеры следующей лексемы будут *добавлены* к текущему содержимому yytext.
- yyless(n) — сокращает строку в yytext до n первых литер, *возвращая* остаток во входной поток.

Листинг 3.7 (ex6.l). Вывод строки наискосок при помощи yyless

```
%%
(.)+    {
          printf(">%s\n", yytext);
          if (yyleng > 1) yyless(yyleng/2);
        }
%%
```

Низкоуровневый ввод-вывод

Пользователь может обращаться к функциям низкоуровневого ввода-вывода, которые используются лексическим анализатором:

- input — чтение следующей литеры из входного потока (в конце потока считывается null-литера);
- output(c) — запись литеры **c** в выходной поток;
- unput(c) — запись литеры **c** во *входной* поток.

В следующем примере функция input используется для поиска конца комментария, заданного в стиле языка C — /* */. Также демонстрируются макроопределения 16-ричных цифр H, десятичных цифр D и букв L и их подстановки: {H}, {D} и {L}.

Листинг 3.8 (ex7_1.l). Макросы и ввод-вывод низкого уровня

```
D   [0-9]
H   [0-9A-Fa-f]
L   [_A-Za-z]

%%
{L}({L}|{D})*   printf( "ident: %s\n", yytext );
0{H}+(H|h)?     |
{D}{H}*(H|h)    printf( "hex: %s\n", yytext );
{D}+            printf( "decimal: %s\n", yytext );
"/*"            skip_comments();
.               ;
%%

void skip_comments()
{
    int c = '*';     /* int, not char! */


    while( c != '/' ) {
        while( input() != '*' );
            c = input();
        if( c != '/' )
            unput (c);
    }
}
```

В примере, приведенном в Листинге 3,8, определены правила для распознавания имен и чисел (десятичных и 16-ричных чисел в стиле ассемблера a86[8]). Для сокращения записи этих правил, в разделе определений заданы *макроопределения* шаблонов, обозначающих буквы, десятичные и 16-ричные цифры; подстановки заданы именами макрокоманд в фигурных скобках.

Вызов unput в функции skip_comments предназначен для обработки частного случая /*?**/ (подряд более одной '*' перед '/').

Внимание:

В этом примере нет проверки конца входного потока, так что *незакрытый* комментарий приведет к зацикливанию в процедуре skip_comments. Правильное решение — всегда проверять результат input на равенство EOF, как показано в Листинге 3.9.

[8] Если число начинается с 0, то для a86 оно 16-ричное; например, 010 — это десятичное 16.

Листинг 3.9 (ex7_2.l). Проверка EOF при использовании input

```
void skip_comments()
{
    int c = '*';     /* not char! */


    do {
        while ((c = input()) != '*' && c != EOF) ;
        while ((c = input()) == '*') ;
    } while (c != '/' && c != EOF);
    if (c == EOF) {
        fprintf(stderr, "?-EOF in comment\n");
        exit(1);
    }
}
```

Если представить себе входной поток в виде магнитофонной ленты, то функция input считывает ее при воспроизведении, а unput — это запись на перемотке в начало.

В следующем примере задано реверсирование идентификаторов, начинающихся с '@':

Листинг 3.10 (ex8_1.l). Функция unput

```
    int  i;

%%
\@[A-Za-z]+   {
                  for( i = 1; i < yyleng; i++ )
                      unput( yytext[i] );
              }
%%
```

Этот пример годится не для всех реализаций lex. Возможно, что функция unput будет изменять величину yyleng (уменьшать на 1) и содержимое yytext (удалять крайнюю литеру), что вполне логично. Поэтому лучше использовать копию yytext и yyleng, как показано в следующем примере.

Листинг 3.11 (ex8_2.l). Дублирование yytext и yyleng при работе с unput

```
    int   i, len;
    char  *p;
%%
\@[A-Za-z]+    {
                 len = yyleng;
                 p = (char *)strdup(yytext);

                 for( i = 1; i < len; i++ )
                     unput( p[i] );
               }
%%
```

Управление правилами

Рассмотрим выбор правил при сопоставлении и управление множеством правил.

Разрешение двусмысленностей

Если при поиске лексемы входная последовательность может быть распознана несколькими шаблонами, то набор правил двусмысленный. В этой ситуации правило выбирается по следующей схеме:

- Предпочтение отдается соответствию большей длины;
- Если одна и та же последовательность соответствует нескольким правилам, предпочтение отдается тому правилу, которое задано раньше других.

Листинг 3.12 (ex9.l). Двусмысленный набор правил

```
%%
read    { printf( "operation: " ); ECHO; }
[a-z]+  { printf( "identifier: " ); ECHO; }
%%
```

Ввод "ready" принимается вторым правилом, поскольку "[a-z]+" распознает все 5 литер ("ready"), в то время как первое правило — только 4 ("read"). При вводе "read" оба правила распознают одинаковое число литер — 4, и будет выбрано первое правило, т. к. оно задано *раньше*. Ввод меньшей длины, например, "re," не приводит к неопределенности, поскольку воспринимается только вторым правилом.

Принцип предпочтения соответствия наибольшей длины действителен и для правил с концевым контекстом[9].

Для правил с выражениями типа ".*" поиск наиболее длинного соответствия приводит к неожиданным результатам. Например, для выявления строк в одиночных кавычках может показаться подходящим следующее решение.

Листинг 3.13 (ex10.l). Неправильный шаблон для распознавания строки в кавычках

```
%%
'.*'   ;
%%
```

Но этот шаблон задает поиск самой дальней закрывающей кавычки, хотя и в пределах строки. То есть при вводе

```
'first' here, 'second' there
```

будет выявлено

```
'first' here, 'second'
```

Хорошо, что поиск по шаблону ".*" ограничен текущей входной строкой, т. к. '.' означает любую литеру *кроме* новой строки. Попытка обойти это ограничение с помощью шаблона (.|\n)+ приведет к бесконечному сопоставлению.

Правильное решение формулируется так: между кавычками могут быть любые литеры кроме *кавычки* и конца строки.

Листинг 3.14 (ex11.l). Правильный шаблон для распознавания строки в кавычках

```
%%
'[^'\n]*'   ;
%%
```

Стартовые условия

Стартовые условия позволяют на ходу изменить множество действующих правил — и тем самым приспособиться к изменению контекста.

Но сначала рассмотрим более простой способ приспособиться к изменению контекста: с использованием переменной состояния. Предположим, требуется в каждой строке заменить

[9] При сравнении "хвост" учитывается.

"magic" на "first", "second" или "third' в зависимости от того, какая цифра была в начале строки — 1, 2, или 3.

Листинг 3.15 (ex12.l). Использование переменной состояния

```
    int state;

%%
^1      { state = 1; ECHO; }
^2      { state = 2; ECHO; }
^3      { state = 3; ECHO; }
\n      { state = 0; ECHO; }
magic   { switch (state) {
            case 1: printf("<first>");  break;
            case 2: printf("<second>"); break;
            case 3: printf("<third>");  break;
            default : ECHO;
          }
        }
%%
```

Теперь решим эту задачу при помощи стартовых условий. Чтобы воспользоваться ими, их нужно сначала объявить:

```
%start cond1, cond2, ...
```

или, чуть короче:

```
%s cond1, cond2, ...
```

Эти условия можно добавить к правилам, записав:

```
<cond>шаблон
```

Такое правило действует тогда, когда текущее стартовое условие анализатора = cond. А оно устанавливается макрокомандой:

```
BEGIN (cond);
```

или, без скобок:

```
BEGIN cond;
```

Вернуться к исходному (нулевому) стартовому условию можно так:

```
BEGIN (INITIAL);
```

Правило может быть активным при *нескольких* стартовых условиях, они записываются через запятую:

```
<cond1, .., condN>шаблон
```

Замечания:

- Правила *без* стартового условия активны всегда.
- На уровне реализации стартовые условия — это целые числа (в частности, INITIAL = 0), что позволяет их трассировать, как показано в Листинге 3,17.

Листинг 3.16 (ex13_1.l). Решение при помощи стартовых условий

```
%START c1 c2 c3

%%
^1          { ECHO; BEGIN c1; }
^2          { ECHO; BEGIN c2; }
^3          { ECHO; BEGIN c3; }
\n          { ECHO; BEGIN 0;  }
<c1>magic   printf( "<first>"  );
<c2>magic   printf( "<second>" );
<c3>magic   printf( "<third>"  );
%%
```

Листинг 3.17 (ex13_2.l). Трассировка стартовых условий

```
%{
#define YY_USER_ACTION { fprintf(stderr, "<%d>", YYSTATE); }
%}
```

Макроопределение YY_USER_ACTION, по умолчанию пустое, позволяет задать код, который выполняется перед действием *любого* правила.

Макрокоманда YYSTATE возвращает численное значение текущего стартового условия. Выясните значения стартовых условий в примере.

Действие *REJECT*

Во всех рассмотренных программах выявляются смежные (примыкающие друг к другу) последовательности. Анализ вложенных и перекрывающихся последовательностей требует применения специальных средств.

В следующем примере запрограммирован счет последовательностей "she" и "he". Но эта программа не выявляет экземпляры "he" внутри "she", т. к. после распознавания "she" эти литеры уходят из входной последовательности.

Листинг 3.18 (ex14_1.l). Подсчет количества *she* и *he* без учета *he* внутри *she*

```
    int s = 0, h = 0;
%%
she     s++;
he      h++;
.|\n    ;
%%

main()
{
    while( yylex() );
    printf( "she: %d times, he: %d times\n", s, h );
}
```

Для выявления вложенной последовательности нужно:

1. вернуть принятую последовательность во входной поток;
2. исключить правило, которым была распознана эта последовательность;
3. возобновить сопоставление.

Первая фаза этого действия может быть реализована вызовом yyless(0), вторая — при помощи стартовых условий. Но можно задать это действие одной макрокомандой REJECT: как показано в следующем примере.

Листинг 3.19 (ex14_2.l). Подсчет всех экземпляров *she* и *he*

```
%%
she     { s++; REJECT; }
he      { h++; REJECT; }
.|\n    ;
%%
```

При обнаружении "she" увеличивается счетчик s, команда REJECT отвергает правило и возвращает "she" на вход. Затем предпринимается попытка заново сопоставить тот же ввод с оставшимися шаблонами.

В этом примере можно учесть то, что "she" включает в себя "he", но не наоборот, и убрать REJECT из второго действия. Но когда в шаблонах задано повторение, невозможно предугадать, сколько литер каким правилом будет распознано.

В примере с "she" и "he" можно заменить REJECT на yyless[10].

Листинг 3.20 (ex14_3.l). Подсчет *she* и *he* с использованием yyless

```
%%
she    { s++; yyless(1); }
he     { h++; }
.|\n   ;
%%
```

[10] Оператор REJECT не работает с ключами –f и –F и не поддерживается в ранних реализациях lex — например, в той, что входит в состав ОС QNX 4.22.

Тема 4. Программирование синтаксического разбора на языке *yacc*

Язык yacc (yet another compiler compiler) позволяет описать синтаксический разбор как набор правил, определяющих синтаксическую структуру ввода, с действиями на языке C.

Исходная программа транслируется yacc в модуль на языке C, в котором определена глобальная функция yyparse, реализующая алгоритм синтаксического разбора в соответствии с заданной грамматикой.

Функция yyparse многократно обращается к внешней функции yylex, которая должна возвращать код лексемы в виде целого положительного числа (или 0 в конце ввода). Код лексемы, возвращаемый yylex, может сопровождаться величиной в переменной yylval (т. н. сопутствующее, или семантическое значение). Интерфейс между функциями yyparse и yylex на этапе компиляции устанавливает заголовочный файл y.tab.h, сгенерированный yacc; там содержатся определения кодов лексем и типа переменной yylval.

Функция yyparse возвращает 0, если конец ввода обнаружен тогда, когда входная последовательность лексем соответствует правилу для символа верхнего уровня грамматики (стартовый символ). Ненулевой результат yyparse говорит о синтаксической ошибке: либо входная последовательность не соответствует ни одному из правил, либо в конце ввода не выполнено правило для стартового символа. В этом случае вызывается функция yyerror, которая должна быть, наряду с main, определена пользователем.

Дадим краткий формальный обзор языка yacc. В дальнейшем лексемы называются также *терминальными* символами; а символы, определенные через другие символы (т. е. конструкции из символов), называются *нетерминальными*.

Структура и синтаксис *yacc*-программы

Форма исходного текста yacc-программы полностью совпадает с формой для lex-программы:

```
определения
%%
правила
%%
процедуры пользователя
```

Все, что следует после второго разделителя "%%" (секция процедур), переносится в C-программу без анализа и изменений. В секции правил допускаются комментарии в стиле языка C и включаемый код на языке C в форме:

```
%{
code
%}
```

Особенности секции определений

Объявления, специфические для yacc-программы:

* объявление объединенного типа (поддерживает разные типы сопутствующего значения):

```
%union
{
type_1 name_1;
...
type_n name_n;
}
```

* объявление стартового символа, в форме:

```
%start start_sym
```

* объявления лексем[11]:

```
%token SYM1 SYM2 ...
```

* либо, с уточнением типа сопутствующего значения:

```
%token <type_k> SYM1 SYM2 ...
```

* объявление типа сопутствующего значения для нетерминального символа:

```
%type <type_k> sym1 ...
```

Формат правил и действий

Правила записываются в форме:

```
sym : SEQ ;
```

где sym — имя определяемого нетерминального символа, SEQ — определение символа в виде последовательности имен терминальных и/или нетерминальных символов.

Разделителями символов в списке SEQ являются пробел, табуляция или новая строка. Точка с запятой разделяет правила.

После любого из символов SEQ может быть задано действие — *составной* оператор языка C, т. е. любое число простых операторов внутри фигурных скобок[12].

Через псевдопеременные $1, $2 и т. д. открыт доступ к стеку *семантических* значений, куда помещаются величины, сопутствующие символам. Семантическое значение символа sym доступно через псевдопеременную $$.

Разные определения одного и того же нетерминального символа можно объединить при помощи знака "|". Например:

```
sym :   SEQ_1
    |   SEQ_2
    ;
```

означает

```
sym :   SEQ_1 ;
sym :   SEQ_2 ;
```

Символ может быть определен и в виде пустой последовательности:

```
sym : /* empty */ ;
```

Взаимодействие модулей *lex* и *yacc*

Взаимодействие модулей, написанных на lex и yacc, поясним на примере программы из каталога _date/v1. Эта программа только проверяет структуру ввода.

Листинг 4.1 (v1.y). Простейший синтаксический анализатор на языке yacc

```
%token      NUMBER MONTH
%start      date

%%
date :      MONTH NUMBER NUMBER
%%
```

[11] Лексемы принято записывать с большой буквы, чтобы отличать их от нетерминальных символов.

[12] В данном пособии рассматриваются только действия в конце правил. Действие в середине правила — трюк для опытных пользователей.

В этой спецификации определены лексемы NUMBER и MONTH и задан стартовый символ — date. (Стартовый символ — это один из нетерминальных символов, обнаружение которого представляет цель синтаксического разбора.) Затем следует определение date через три терминальных символа. Точка с запятой в конце определения, отделяющая правила друг от друга, в примере отсутствует, т. к. правило здесь единственное.

Лексический анализ сводится к выявлению чисел и строк с названиями месяцев, что задано следующей lex-спецификацией.

Листинг 4.2 (v1.l). Модуль на языке lex для синтаксического анализатора

```
%{
#include "y.tab.h"
%}

%%
[0-9]+      { return NUMBER; }
jan         |
...
dec         { return MONTH; }
[ \t\n]     ;
.           { return 0; }
%%

#ifndef yywrap
int yywrap () { return 1; }
#endif
```

Если на входе появится литера, не относящаяся к числам и названиям месяцев и не являющаяся разделителем (пробелом, табуляцией или новой строкой), функция yylex вернет ноль — признак конца ввода для yacc-модуля. Имена NUMBER и MONTH — это константы из файла y.tab.h, полученного в результате трансляции v1.y.

Примечание: здесь, в отличие от примеров на тему lex, функция yylex, обнаружив лексему, сразу возвращает ее код — он обрабатывается в вызывающей функции yyparse.

Для получения исполняемой программы вызовите сценарий **build.sh**. В нем задан вызов **lex** для всех файлов с расширением -l из текущего каталога (у нас один — v1.l), вызов **yacc** для модулей с расширением -y и, наконец, вызов **cc** для всех модулей на языке C, а именно: v1.c (результат трансляции v1.l), y.tab.c (результат трансляции v1.y) и zz.c. Последний играет ту же роль, что yy.c в примерах на тему lex: он содержит определение функции main, которая вызывает функцию синтаксического разбора yyparse. Также в zz.c определена функция yyerror(char *) — функция yyparse вызовет ее при синтаксической ошибке, с указателем на строку "syntax error". Здесь же определена переменная yydebug, для включения режима отладки.

Проверьте полученную программу, задав ей на входе **<test.in**. Изменив на время опыта структуру test.in (например, добавив еще одно число перед знаком '!'), проверьте реакцию программы.

Трассировка правил

В модуле zz.c можно включить режим трассировки, т. е. вывод правил, применяемых при разборе. Для этого нужно в определении переменной yydebug исправить 0 на 1 и заново выполнить компиляцию. Проверьте. Программа ex1 ничего кроме трассы не выводит, но в дальнейшем, чтобы потоки stdout и stderr не смешивались, задавайте перенаправление хотя бы для одного из них: **>test.out** и/или **2>test.err**.

В сообщениях трассировки **shift** означает продолжение разбора с переходом в другое состояние, а **reduce** — свертку последовательности символов, замену ее одним символом в

результате применения правила. В ходе разбора автомат меняет свои состояния (state); возможные состояния перечислены в файле y.output. Эти вопросы рассматриваются более подробно при обсуждении листинга 4.15, а пока можно обойтись без трассировки.

Литеральные лексемы

Из заголовочного файла y.tab.h видно, что коды терминальных символов, определенных при помощи ключевого слова %token, начинаются с 257. Код 0 зарезервирован для признака конца ввода, а коды от 1 до 256 — для литеральных лексем, или "литералов".

Использование литералов иллюстрируется примером из каталога _date/v2.

В определении date появилась запятая в одиночных кавычках — это и есть литерал, то есть терминальный символ, код которого равен ASCII-коду запятой.

Листинг 4.3 (v2.y). Литерал в определении нетерминального символа

```
date :   MONTH NUMBER ',' NUMBER
```

В предыдущем примере функция yylex лексического анализатора при чтении запятой возвращала результат 0. Теперь в lex-модуль добавлено правило, которое в этом случае возвращает код запятой.

Листинг 4.4 (v2.l). Передача литерала из lex-модуля

```
","      { return yytext[0]; }
```

Протестируйте эту программу. Какова теперь допустимая структура ввода? Измените программу так, чтобы можно было бы использовать запятую и точку с запятой.

Сопутствующие значения

Если бы лексический анализатор вычислял *величины* месяцев и чисел и передавал их вместе с кодом лексемы, то синтаксический анализатор мог бы выводить дату и проверять ее допустимость. В примере _date/v3 эти возможности использованы.

Листинг 4.5 (v3.l). Задание типа и величины сопутствующего значения

```
%{
#include <stdlib.h>
#include "y.tab.h"

#define YYSTYPE int
extern YYSTYPE yylval;
%}

%%
[0-9]+      { yylval = atoi(yytext); return NUMBER; }
jan         { yylval =  0; return MONTH; }
feb         { yylval =  1; return MONTH; }
...
dec         { yylval = 11; return MONTH; }
","         { return yytext[0]; }
[ \t\n]     ;
.           { return 0; }
%%
```

Здесь добавлено определение типа сопутствующего значения YYSTYPE и ссылка на внешнюю переменную yylval. Лексеме NUMBER сопутствует значение десятичного числа, а лексеме MONTH — номер месяца в диапазоне [0..11].

Синтаксический анализатор использует сопутствующие значения следующим образом. Когда yylex возвращает управление yyparse, величина yylval записывается в стек значений; так продолжается, пока правило не будет применено. Доступ к этим значениям открыт через псевдопеременные $n. В начале кадра стека оставлено место для сопутствующего значения *определяемого* символа (псевдопеременная $$).

Листинг 4.6 (v3a.y). Доступ к семантическим значениям

```
%%
date :  MONTH NUMBER ',' NUMBER
            { printf("m-d-y: %2u-%2u-%4u\n", $1+1, $2, $4); }
%%
```

Семантическое значение первого символа доступно через $1 — это номер месяца от 0 до 11, а $2 и $4 — значения дня и года. Литерал ',' в третьей позиции тоже считается символом; у него тоже есть значение, доступное через $3 — но там сейчас случайная величина, так как функция yylex, обнаружив запятую, в yylval ничего не записала.

Листинг 4.7 (v3b.y). Проверка даты и вывод количества дней от 1970 г.

```
%{
long abs_date (int, int, int); /* month (0-11), day, year */
%}

%token  NUMBER MONTH
%start  date

%%
date :  MONTH NUMBER ',' NUMBER
            { printf("%ld\n", abs_date($1, $2, $4)); }
%%
```

Проверка даты и вычисление количества дней, прошедших от 01/01/1970 выполняется в функции abs_date (см. модуль abs_date.c) при помощи библиотечной функции mktime. Для проверки даты пригодилось то, что mktime корректирует значения за пределами диапазона составляющих даты-времени (например, 31/02 или 24:00:01)[13].

Значение числа дней можно было бы использовать в качестве сопутствующего значения для символа date. В следующем примере семантическое значение date формируется в конце правила для date и используется в правиле для between. Семантическое значение between не формируется за ненадобностью.

Листинг 4.8 (v3c.y). Семантическое значение date и вычисление разницы между датами

```
%token  NUMBER MONTH
%start  between

%%
date :    MONTH NUMBER ',' NUMBER
            { $$ = abs_date($1, $2, $4); }
between : date '-' date
            { printf("%ld\n", $1 - $3); }
%%
```

Замечание: Величина $$ изначально равна величине $1; можно считать, что присвоение $$ = $1 — это действие по умолчанию.

[13] Хотя POSIX не рекомендует пользоваться этим свойством mktime [2]

Пример в каталоге _date/v3/c некорректный в том смысле, что тип у сопутствующих значений — int, а у функции abs_date — long. Поэтому при присвоении $$ = abs_date(...) отбрасывается старшая часть результата. Можно выйти из положения, задав тип long для всех сопутствующих значений. Пример приведен в каталоге _date/v3/d, а мы рассмотрим другой вариант.

Сопутствующие значения разных типов

Иногда требуется возвращать сопутствующие значения разных типов, например, int и char*, притом что канал передачи значений от yylex к yyparse единственный — переменная yylval. В этом случае используется объединение (union).

Рассмотрим примеры из _date/v4.

Листинг 4.9. Определение сопутствующего значения нескольких типов

```
%union
{
    int    ival;
    char * text;
};
```

Выполните пример в каталоге v4/a. Трансляция yacc-модуля не прошла, поскольку в нем не задана информация о типе $1, $2 и $4 — ведь теперь у сопутствующего значения не один тип, а два.

Тип можно указать явно при обращении к $-переменной.

Листинг 4.10 (v4b.y). Явное указание типа при обращении к $-переменной

```
date :   MONTH NUMBER ',' NUMBER
            { print($<text>1, $<ival>2, $<ival>4); }
```

Тип может быть указан и при *объявлении* терминального символа. Тогда при обращении к $-переменным его не придется уточнять, и этот вариант предпочтительный.

Листинг 4.11 (v4c.y). Задание типа при объявлении символа

```
%token   <ival> NUMBER
%token   <text> MONTH

%%
date :   MONTH NUMBER ',' NUMBER
            { print($1, $2, $4); }
%%
```

В lex-модуле мы обращаемся к yylval как к варианту union в языке C.

Листинг 4.12 (v4.l). Формирование сопутствующего значения в lex-модуле

```
[0-9]+      { yylval.ival = atoi(yytext); return NUMBER; }
jan         |
feb         |
...
nov         |
dec         { yylval.text = strdup(yytext); return MONTH; }
```

В результате использования %union определение YYSTYPE (в форме C-объединения) попадает в заголовочный файл y.tab.h. Это определение не нужно дублировать в lex-модуле, достаточно директивы #include "t.tab.h".

Замечание:

При формировании указателя строки использована библиотечная функция strdup, копирующая содержимое yytext в динамическую память. Передача ссылки непосредственно на yytext (yylval.text = &yytext[0]) была бы ошибкой, т. к. к моменту использования этой ссылки (функцией print) содержимое yytext уже изменится — там будут цифры.

Вернемся к примеру, где подсчитывается количество дней между двумя датами. В нем сопутствующие значения должны быть двух типов:

- int — для месяца, дня и года;
- long — для нетерминального символа date (количество дней от 01/01/1970).

Листинг 4.13 (v5.y). Вычисление количества дней между двумя датами

```
%union
{
    int   ival;
    long  lval;
};

%token   <ival> NUMBER MONTH
%type    <lval> date
%start   between

%%
date :    MONTH NUMBER ',' NUMBER
            { $$ = abs_date($1, $2, $4); }
between : date '-' date
            { printf("%ld\n", $1 - $3); }
%%
```

Разрешение двусмысленностей

Если некая входная последовательность может быть распознана сразу несколькими шаблонами, то набор правил двусмысленный.

Транслятор yacc в этих случаях выводит предупреждение:

- shift/reduce conflict — выбор между применением правила (reduce) и продолжением разбора (shift) в соответствии с другим правилом.
- reduce/reduce conflict — выбор между применением нескольких правил.

Правило выбирается по схеме, напоминающей ту, что принята в lex:

- предпочтение отдается соответствию большей длины, т. е. столкновение shift/reduce разрешается в пользу shift.
- если одна и та же последовательность соответствует нескольким правилам (конфликт reduce/reduce), предпочтение отдается тому правилу, которое задано раньше других.

Рекурсивные правила

Обратимся к программе в каталоге list/v0. Она разбирает список чисел, разделенных запятыми, и выводит число элементов в списке.

Листинг 4.14 (v0/c1.l). Лексический анализатор для разбора списка чисел

```
%%
[0-9]+      { yylval = atoi(yytext); return NUM; }
(.|\n)      return yytext[0];
%%
```

Лексический анализатор в Листинге 4,14 распознает десятичные числа и передает их синтаксическому анализатору в виде лексемы NUM с сопутствующим значением yylval. Все прочие литеры он передает в yacc-модуль в виде литералов.

Листинг 4.15 (v0/c1.y). Синтаксический анализатор для разбора списка чисел

```
%start __list
%%
__list: _list       { printf("No. of items: %d\n", $1); }

_list: /* empty */  { $$ = 0; /* size is 0 */ }
      | list        /* not empty, $$ == $1 by default */
      ;

list: NUM           { $$ = 1; }       /* size := 1 */
    | NUM ',' list  { $$ = $3 + 1; }  /* size := size of sublist + 1 */
    ;
%%
```

Скомпилируйте программу c1 и подайте на ее вход: 1,2,3<**Enter**><**Ctrl+D**><**Enter**>.

Последует сообщение "?-syntax error". Чтобы выяснить причину, включите трассировку; для этого в файле zz.c исправьте в определении yydebug 0 на 1 и повторите трансляцию, задав при вызове yacc ключи **–vtd**.

При вызове c1 рекомендуется перенаправить вывод: **c1 >test.out 2>test.err**. Трассировка попадет в отдельный файл test.err, не смешиваясь с выводом программы c1.

Задав на входе 1,2,3<**Enter**>, получим в test.err следующее:

```
yydebug: state 0, reading 257 (NUM)
yydebug: state 0, shifting to state 1
yydebug: state 1, reading 44 (',')
yydebug: state 1, shifting to state 5
yydebug: state 5, reading 257 (NUM)
yydebug: state 5, shifting to state 1
yydebug: state 1, reading 44 (',')
yydebug: state 1, shifting to state 5
yydebug: state 5, reading 257 (NUM)
yydebug: state 5, shifting to state 1
yydebug: state 1, reading 10 (illegal-symbol)
yydebug: error recovery discarding state 1
...
```

В каждой строке, пока не появилась ошибка, показан номер состояния конечного автомата при синтаксическом разборе. Что значат эти номера и состояния, можно выяснить в файле y.output, полученном при трансляции yacc-модуля.

Ниже приведен фрагмент файла y.output для рассматриваемого примера.

```
(1)         0  $accept : __list $end
(2)         1  __list : _list
(3)         2  _list :
(4)         3        | list
(5)         4  list : NUM
(6)         5       | NUM ',' list
(7)     state 0
(8)             $accept : . __list $end  (0)
(9)             _list : .  (2)
(10)            NUM  shift 1
```

```
(11)            $end  reduce 2
(12)            __list  goto 2
(13)            _list  goto 3
(14)            list  goto 4
                ...
        4 terminals, 4 nonterminals
        6 grammar rules, 7 states
```

В строках (1–6) перечислены правила из yacc-модуля. Дальше идет описание состояний. Работа автомата начинается из состояния 0. В каждом состоянии у автомата могут быть, в общем случае, несколько альтернативных целей, и выбор зависит от очередного символа.

Текущий пункт на пути к цели отмечается точкой. Например, в состоянии 0 автомат должен получить либо символ __list согласно (8), либо конец ввода согласно (9).

После целей (8–9) перечислены ожидаемые (допустимые) символы и реакция на них. Так, запись в строке (10) означает: при получении лексемы NUM перейти в состояние 1. Слово shift означает переключение состояния с накоплением данных в стеке. Действительно, одно число рано считать списком — за ним могут следовать, через запятую, другие числа.

Операции goto переключают состояние без накопления данных.

Операция reduce означает применение правила, с удалением данных из стека. Например, согласно (11), конец ввода в состоянии 0 приведет к применению правила 2. Это правило, согласно (3), относится к пустому списку.

Вернемся к трассе программы при вводе 1,2,3<**Enter**>. Читаем: в состоянии 0 получен код 257, что соответствует лексеме NUM; в результате перешли в состояние 1. Дальше, в состоянии 1 получен код 44, что соответствует ASCII-коду ',' (*Приложение 5*) и т. д. — до получения символа 10, недопустимого в состоянии 1. Код 10, по таблице ASCII, означает конец строки — литерал '\n'.

Литерал '\n' пришел из lex-модуля. Исправить ситуацию можно двумя способами.

Листинг 4.16 (v0/c2.l). Удаление '\n' при лексическом разборе

```
%%
[0-9]+  { yylval = atoi(yytext); return NUM; }
\n      ;
.       return yytext[0];
%%
```

Листинг 4.17 (v0/c2.y). Включение '\n' в синтаксический разбор

```
__list: _list '\n'  { printf("No. of items: %d\n", $1); }
```

Проверьте эти варианты, собрав программу в сочетаниях: c1.l + c2.y и c2.l + c1.y.

Теперь выясним, как программа реагирует на разделители. Подайте на вход список чисел с *пробелами*: 1 , 2, 5<**Enter**>. Сбой происходит на литере с кодом 32 — то есть как раз на пробеле. Фильтрацию пробелов и табуляций имеет смысл выполнять в lex-модуле.

Листинг 4.18 (v0/c3.l). Удаление разделителей при лексическом разборе

```
%%
[0-9]+      { yylval = atoi(yytext); return NUM; }
[ \t\n]+    ;
.           return yytext[0];
%%
```

А вот пример, как это не надо делать.

Листинг 4.19. Ошибка: включение разделителей в лексему

```
[ \t\n]*[0-9]+[ \t\n]*  { yylval = atoi(yytext); return NUM; }
```

Почему не надо включать разделители в шаблоны лексем? Выглядит громоздко, и, что еще хуже, разделители попадут в yytext и тогда для вычисления семантического значения придется от них избавляться — теперь уже средствами C.

В каталоге list/v1 к разбору списка добавлен вывод элементов.

В описании непустого списка в c1.y используется правая рекурсия, а в c2.y — левая. При левой рекурсии применение правила откладывается до конца списка, что требует больше ресурсов и может привести к исчерпанию памяти. Убедитесь, что c1 и c2 выводят элементы списка в разном порядке. В каком варианте список выводится от начала к концу?

Список источников

1. *Цыган, В.Н.* Транслирующие системы. Санкт-Петербург, СПбПУ, 2014. <URL:http://dl.unilib.neva.ru/dl/2/3981.pdf>
2. *Donald A. Lewine*. POSIX Programmer's Guide. O'Reilly & Associates, 1991, 611 p.
3. *John R. Levine*. Lex & Yacc / John R. Levine, Tony Mason, Doug Brown. O'Reilly & Associates, 2nd ed., 1992, 366 p.
4. *Andrew W. Appel*. Modern Compiler Implementation in C / Andrew W. Appel, Maia Ginsburg. Cambridge University Press, 1998, 560 p.
5. *John Levine*. Flex & Bison: Text Processing Tools. O'Reilly Media, 2009, 292 p.
6. *Terence Parr*. The Definitive ANTLR 4 Reference. O'Reilly, Pragmatic Bookshelf, 2nd ed., 2013, 328 p.

Приложение 1. Варианты заданий

Второе задание по теме 1

Программа выполняет функции базового сканера из c/scanner, то есть распознает идентификаторы, числа и литералы; иными словами, она *дополняет* функции базового сканера. Вывод — в стиле базового сканера, с использованием функции prn_token.

Замечания:

- В тестовый набор должны входить, как минимум, все ситуации, отмеченные в заданиях,
- При обработке чисел надо тестировать арифметическое переполнение.
- В задачах с 16-ричными константами используйте макрос isxdigit.
- В задачах, где надо удалить комментарии, не нужно их выводить и тем более не следует относить их к лексемам. Комментарии должны исчезнуть, как будто их не было. Ваш сканер отличается от базового только тем, что умеет пропускать комментарии.

Внимание: Второе задание по теме 1 является также первым заданием по теме 3. Поэтому тестовый набор для программы на C должен быть применим, без изменений, к программе на Lex. Программа на Lex разрабатывается в том же каталоге, что и программа на C; а для тестирования используется тот же сценарий **x.sh** и те же файлы **-in** и **-out**.

1. 16-ричные константы в стиле C, например, 0x1fa2. Ввод “0x=” распадается на число 0, идентификатор x и знак равенства, а ввод 01fe – на число 01 и идентификатор fe.

2. 16-ричные константы в стиле a86, сразу в трех вариантах (все в одном сканере). В a86 16-ричное число задается либо нулем в начале (030 означает десятичное 48), либо суффиксом h или xh. Примеры: 01fa2, 3fh, 8fxh. Ввод “1fz” распадается на две лексемы: число 1 и идентификатор fz.

3. 16-ричные константы в стиле Modula-2: с ведущим 0 перед цифрой a-f и обязательным суффиксом h. Примеры: 0fah, 7fah. Ввод “1fz” распадается на две лексемы: число 1 и идентификатор fz.

4. 16-ричные, 8-ричные и двоичные константы в стиле языка Step 7 (все в одном сканере). Примеры: 16#fa, 8#177, 2#10101. Ввод “2#3” распадается на три лексемы: число 2, знак # и число 3. Аналогично для 16-ричных и 8-ричных. Размерность констант — в пределах 32 бит.

5. 16-ричные и двоичные константы в стиле языка Ada (все в одном сканере). Примеры: 2#1110101111 или 2#11_1010_1111, 16#abcded или 16#ab_cdef. Подчеркивание, если оно используется, должно идти через каждые 4 цифры, считая от младших. Ввод 2#11_111 распадается на 2#11, литеру подчеркивания и десятичное число 111.

6. Знаковые числа с фиксированной точкой, например: -8.234. Ввод “1.x” или “x.23” распадается на три лексемы: число, точка и идентификатор. Предусмотрите проверку переполнения[14].

7. Пропускать комментарии в стиле языка C, то есть /* … **/. Если ввод заканчивается до закрывающей скобки “*/” — это незавершенный комментарий, ошибка. Следует также проверить вариант с лишней звездочкой: /* … **/. Сами комментарии выводить не надо.

8. Пропускать текст между #if 0 и #endif. Таким способом в C исключают фрагменты текста, причем, в отличие от /*... */, эти комментарии могут быть *вложенными*. Если #endif отсутствует, это ошибка. Закомментированный текст выводить не надо.

9. Пропускать *вложенные* комментарии в стиле языка Modula-2: (*... (*…*) ...*). Если в конце ввода комментарий не закрыт, это ошибка. Комментарии выводить не надо.

[14] Функция **strtod** также устанавливают errno == ERANGE, подобно **strtoll** в базовом сканере.

10. *Многострочный* комментарий в стиле ассемблера для i80x86: от слова COMMENT до знака, заданного после COMMENT (например, “COMMENT $ ... $”). Внимание, возможны три разных случая: COMMENT x — здесь литера x считается ограничителем комментария (между COMMENT и первым 'x' — сколько угодно пробелов и табуляций); COMMENT$ — доллар является ограничителем; COMMENTx — вообще не комментарий, а идентификатор. Текст комментария выводить не надо.

11. Десятичные константы со знаком, включая "длинные целые" в стиле Step 7. Примеры: -1, 1, +99, **L#**-1, **L#**14. Значение "длинной" константы должно быть представлено 32 битами, а "короткой" — 16. Например, L#-1 — это 0xffffffff, а -1 — 0xffff (так и следует их выводить). Ввод “L#f3” распадается на три лексемы: идентификатор L, знак # и идентификатор f3. Надо предусмотреть проверки на переполнение — отдельно для 16- и 32-битных значений.

12. Шестнадцатеричные константы в стиле Step 7 всех размерностей: байт, слово и двойное слово. Примеры: B#16#ff, W#16#1fe, DW#16#7fffffff. Предусмотреть проверки переполнения для байт, слов и двойных слов. Например, B#16#00f05 считается ошибкой, т. к. 16-ричное число f05 — вне диапазона байта.

13. Десятичные константы со знаком, включая "длинные целые" в стиле C. Примеры: -1, 1, +99, -1**L**, 14**L**. Длинные константы должны быть представлены 32 битами, а короткие — 16. То есть, -1L — это 0xffffffff, а -1 — 0xffff (так их и следует выводить). Надо предусмотреть разные проверки на переполнение: для коротких (16-битных) и длинных (32-битных) значений.

14. Имена, начинающиеся с буквы, за которой следует любое число букв, цифр и символов '_'. Это *взамен* идентификаторов в примере. Также требуется распознавать локальные имена в стиле a86: буква, за которой следует не менее одной цифры, например, m12, z2.

15. Имена, состоящие из любого набора букв и знаков '$', '_', '@', при условии, что хотя бы одна буква должна быть. Ввод @_$$$x — это допустимое имя, но @_$$$1 распадается на @, _, три доллара и единицу. Это *взамен* идентификаторов в примере.

Задания по темам 2 и 4

1. На входе задана директива инициализации массива в стиле языка Step 7, например:

26, 3 (1, 2 (5, 6, 7)), 0

Это список целых 16-битных значений, разделенных запятыми; число перед скобками задает количество повторов списка в скобках. Ошибками считаются: нулевое количество повторов и пустой список в скобках. Постройте программу, которая вычисляет число элементов.

Дополнительное задание: вывести значения элементов через запятую, без скобок:

26, 1, 5, 6, 7, 1, 5, 6, 7, 1, 5, 6, 7, 0

2. Точка в пространстве задана координатами в скобках, например, (1, -2, 16). Любое из чисел может быть опущено, если координата 0. Например, (1,,2) означает (1, 0, 2), а (1, 4) — это (1, 4, 0); начало координат может быть задано даже как (,,) или ().

```
(-1,    ,6) (1, 3, 99)(,-8,-6)
(,,100)
(0, 99)    ()
```

Сделайте программу, которая выведет точки построчно, отобразив *все* координаты.

Замечание: при решении в yacc не надо перечислять все возможные сочетания: (NUM,,), (,NUM,NUM) и т. д., а лучше задать одно правило: (item, item, item), где item — это NUM или ничто.

3–6. На входе — список точек на плоскости. Каждая точка задана парой координат в скобках, например, (-3, 6). Обе координаты должны быть заданы явно, т. е. запись (-5,) ошибочна. Вот как могут быть заданы точки с координатами (1, -5), (10, 6) и (0, -7):

```
(1      ,-5)(10, 6
) (0  ,-7 )
```

Варианты:

- найти точку, наиболее/наименее удаленную от начала координат, вывести дальность и порядковый номер этой точки;
- вывести периметр многоугольника, заданного точками; считаем, что многоугольник замкнутый, то есть недостающая сторона — это отрезок между последней и первой точками;
- убедиться, что точки следуют в порядке удаления от начала координат;
- убедиться, что расстояние между соседними точками становится все больше.

Замечание: для сравнения расстояний не надо вычислять корень — достаточно оценивать сумму квадратов; а корень берется один раз при выводе результата.

7. На входе задан распорядок на один день, который выглядит, например, так:

```
9:30 12:00 Wake up
13:20 15:50     Have a little something
16:00      18:02 Doing Nothing
18:00 23:59 Dinner at English Club
```

Перекрытие интервалов не считается ошибкой, но время 24:00 или 01:60 и т. п. — ошибка. Требуется найти наибольшее "окно" в промежутке от 9:00 до 17:00.

Примечания:

- Время в формате hh:mm пишется слитно, это *одна* лексема с сопутствующим значением, равным числу секунд или минут, прошедших с полуночи.
- Для преобразования времени из формата hh:mm есть стандартная функция mktime. Она принимает структуру с полями tm_hour, tm_min, tm_sec и т. п., а возвращает секунды (см., например, yacc/_date/v5/abs_date.c). Ошибку в исходных данных можно обнаружить, сравнив значения структуры до и после вызова mktime. Например, 22:59:61 после mktime превратится в 23:00:01.

8. На входе задан фрагмент управляющей программы (УП) для системы числового программного управления (СЧПУ), например:

```
N105G1X10
N102X10Y10G0
X-25 G01 Y-5
```

Управляющая программ состоит из кадров, разделенных символом '\n'. Буква и число пишутся слитно, но сами эти пары можно разделять пробелами. Номер кадра N необязателен. Обязательны: подготовительная функция G1 (линейная интерполяция) или G0 (позиционирование), а также *приращения* по координатам X и/или Y. Элемент кадра не может быть задан дважды, кадры N99G0**X**12Y10**X**8 и **N**1G0**N**2X300Y-20 неправильные.

Полагая, что движение начинается из точки (0, 0), выведите вектора, по которым идет *интерполяция*. В примере: (0, 0) -> (10, 0) и (20, 10) -> (-5, 5).
Примечание: При реализации в уасс сопутствующее значение должно быть сложного типа и должно включать в себя два значения: ASCII-кода буквы и числа, следующего за буквой. Причем ни буква, ни число не являются лексемами! Лексема — это пара буква-число.

9. На входе задан список кадров, как в задании 8. Требуется разобрать список, не вкладывая никакого смысла в буквы, и вывести для каждого кадра перечень пар буква-число в *алфавитном* порядке. Допустимы все буквы от A до Z, например: A2 N1 X20 B-10 Z06 T2. Но ни одна буква не должна повторяться в пределах кадра. Вот примеры ошибочных кадров: X2 C-8 X1, X Z2, G1 99, T 2. См. также *Примечание* к заданию 8.

10. На входе задана директива db (define bytes) распределения байт, в стиле ассемблера для i80x86 например:

```
db      10, 3 dup (?, 4, 18 dup (0), 9), “None”, 2 dup ('Letters', 7)
```

Оператор **n dup (list)** означает n повторений list. "None" — это 'n', 'o', 'n', 'e'. Знак вопроса — это любое значение (можно считать, что это 0). Разработайте программу, которая определяет количество байт, зарезервированных директивой db. Чтобы проверить результат, выполните трансляцию входного файла: **a86 file.in** — и оцените длину полученного com-файла.

Дополнительное задание: сформировать образ данных и вывести его — или в двоичном виде в файл, или в текстовом виде через запятую.

11. На входе задана директива dw (define words) распределения 16-битных слов, в стиле ассемблера a86, например:

```
dw      10, 1k dup (?, 4, 18 dup 0, 2k)
```

В отличие от директивы db в задании 10, в dw не допускаются строки в кавычках.

Особенности a86:

- одиночный элемент в конструкции dup может быть указан без скобок: 18 dup 0;
- суффикс k при числе — это множитель 1024 (то есть 2k означает 2048).

Дополнительное задание: сформировать образ данных и вывести его — или в двоичном виде в файл, или в текстовом виде через запятую.

12. Разберите объявление процедуры в языке Pascal, например:

```
procedure sample (var a, b : real; c : real; var d: boolean; e: char)
```

Результат разбора — число байт, занятых параметрами. Параметры типа char и boolean занимают по 1 байту, integer — 2, real — 4. Параметры, передаваемые по ссылке (они заданы после слова var), занимают по 2 байта независимо от типа данных. В примере результат равен 11: a и b — по 2 байта (var), c — 4 (real), d — 2 (var), e — 1 (char).

13. Преобразуйте макрокоманду push ассемблера a86 в обозначения машинных команд. В качестве операндов push допустимы обозначения 16-битных регистров общего назначения ax, bx, cx, dx, si, di, bp, sp, сегментных регистров ds и es, десятичных чисел со знаком; push без операндов тоже допускается. Примеры операторов push и их преобразование:

```
push ax, bx, 1  -> push ax
```

```
                  push bx
                  push 1
push           -> add sp, 2
```

Дополнительное задание: сделайте сканер нечувствительным к регистру имен (как принято в языках ассемблера); для этого при разборе преобразуйте буквы к верхнему регистру.

14. Разберите объявление С-процедуры, например:

```
void sample (float* a, float *b, float c, char * d, char e)
```

Результат разбора — суммарный объем параметров в байтах, с учетом следующих соглашений: параметр типа char занимает 1 байт, short — 2 байта, long — 4, float — 4, double — 8. В предположении, что разрядность целевого процессора — 16 бит, параметр типа int и параметр-указатель (например, char *) занимают по 2 байта. В примере параметры занимают 11 байт: a — 2 (*), b — 2 (*), c — 4 (float), d — 2 (*), e — 1 (char).

15. Преобразуйте макрокоманду mov ассемблера a86 в команды стандартного ассемблера. В качестве операндов допустимы обозначения: 16-битных регистров общего назначения (ax, bx, cx, dx, si, di, bp, sp), сегментных регистров ds и es, а также десятичное число со знаком (только в последнем операнде). Копирование по команде mov выполняется "справа налево". Примеры операторов и их преобразование:

```
mov ax, bx, 1   -> mov bx, 1
                   mov ax, bx
mov ds, 1       -> push 1
                   pop  ds
~~mov ax, 1, bx~~
```

Дополнительное задание: сделайте сканер нечувствительным к регистру имен (как принято в языках ассемблера); для этого при разборе преобразуйте буквы к верхнему регистру.

16. Разберите определение массива целых чисел на языке С и выведите массив поэлементно. Результат разбора — вывод массива по элементам. Примеры определений:

```
int x[] = { 1, 2, 3 };
int *x = { 1, 2 };
int x[3];                       // 0, 0, 0
int x[5] = { 1, 2, 3 };         // 1, 2, 3, 0, 0
~~int x[2] = { 1, 2, 3 };~~
```

Последний оператор неправильный, потому что в фигурных скобках чисел больше, нежели вмещает массив.

17. Разберите определение массива символов на языке С и выведите массив поэлементно. Результат разбора — вывод массива по элементам. Примеры определений:

```
char s[] = { '1', '2', 3 };
char *p = "Hi!";
char s[3];                      // 0, 0, 0
char s[5] = { 1, 2, 3 };        // 1, 2, 3, 0, 0
~~char s[2] = "Hi!";~~
```

В списках инициализации допускаются не только литеры в одиночных кавычках, но и целые числа. Строка в кавычках занимает в памяти на 1 байт больше, т, к. она автоматически

дополняется нулем. Последний оператор неправильный, т. к. длина строки больше, чем вмещает массив из двух элементов.
18–20. Проверьте файл субтитров (см. texts/subscenes.srt) на ошибки. Если таковых нет, внесите их сами. Ниже, с указанием номера варианта, перечислены правила, соблюдение которых надо проверить:

18) Интервалы времени не пересекаются и идут по возрастающей. То есть, время окончания субтитра > времени его начала и <= времени начала следующего.
19) Текст субтитра умещается в одну или две строки, не больше. Длительность субтитров — в пределах от 0,5 до 7 сек.
20) Каждая из строк содержит не более 40 литер, но и не может быть пустой. Для каждого субтитра скорость чтения текста не превышает 20 знаков в секунду.

Второе задание по теме 3

Если номер вашего варианта [12..23], ищите свое задание в Приложении 3, табл. П3.2.

Внимание: первое задание по Lex —. это второе задание по теме 1,

1. Удалять все пробелы и табуляции в конце строк. Конец строки обозначается или '\n', или специальным символом $. Проверьте оба варианта, каждый на трех входных файлах: у первого в последней строке только пробелы и табуляции, у второго ничего (то есть он заканчивается сразу после перевода строки), а третий заканчивается без перевода строки (то есть последняя строка не завершена).

Примечание: Текстовые редакторы в Unix не допускают висячей строки в конце текста, добавляя в конец файла перевод строки. Поэтому создать третий входной файл можно не иначе как в Windows или с использованием **hexedit**.

2. Выявлять во входном потоке идентификаторы длиной не более 4 литер и выводить их по одному на строке (все прочее — не выводить).

Варианты по возрастанию сложности:

- шаблон для выявления идентификаторов задан без ограничений длины, но в действии проверяем длину идентификатора и выводим только короткие.
- ограничение на длину задаем в шаблоне; в этом случае должно быть также правило для идентификаторов произвольной длины, иначе длинный идентификатор будет разделен на несколько коротких.

Замечание: выводить первые 4 литеры длинного идентификатора — неправильно; длинные идентификаторы вообще не выводятся: ни целиком, ни частично.

3. Заменить знаки табуляций рядами пробелов так, чтобы форматирование текста осталось прежним. Для удобства отладки вместо пробелов выводите "+". При обнаружении литеры табуляции в выходной поток нужно передать пробелы, а их число зависит от текущей позиции в строке. Т. е., для решения задачи необходимо вести *счет* символов в строке.

4. Заменить подряд идущие пробелы табуляциями (с добавлением пробелов в конце) так, чтобы форматирование текста осталось прежним. Для удобства отладки вместо пробелов используйте "видимые" литеры, например, '+'. Пример входного текста:

```
Part+1++++++++++++++++++++Lex
1.+Chapter+1++++++++++++++++++Regular+expressions
12.+Chapter+12++++++++++++++++Implementations
```

Текст после преобразования (в первой строке показаны позиции табуляций):

```
        |       |       |       |       |       |       |
Part+1                  +++Lex
1.+Chapter+1                    Regular expressions
12.+Chapter+12                  +Implementations
```

В первой строке между "Part 1" и "Lex" — 3 табуляции и 3 пробела. Цепочка пробелов во второй строке полностью преобразована в табуляции, поскольку фраза "Regular expressions" начинается как раз в позиции табуляции. Кстати, "expression" тоже начинается в позиции табуляции, поэтому одиночный пробел перед expression заменен на табуляцию (хотя можно было оставить и так). Чтобы выяснить число табуляций и пробелов, нужно знать, в какой позиции начинаются пробелы и где они заканчиваются — для этого нужен *счетчик* литер в строке.

5. Выявить константы ассемблера a86 и вывести их в десятичном формате[15]. Предусмотреть контроль переполнения. Примеры констант:

- 12, -12, +56 — десятичные;
- 177xq, 164q — восьмеричные;
- **1011b**, 011xb — двоичные;
- 012, **011b**, 0fa1, 0axh, 1eh — шестнадцатеричные.

Замечание: Обратите внимание на константы 1011b и 011b — первая из них двоичная, а вторая 16-ричная. Это потому, что в a86 ведущий ноль означает 16-ричное число[16], а буква 'b' этому не противоречит. Иными словами, буква 'b' в 011b — это 16-ричная цифра, а не суффикс двоичного числа.

6. Выявить обозначения регистров процессора i8086: ax, bx, cx, dx, ah, al, bh, bl, ch, cl, dh, dl, si, di, bp, sp, ds, es, cs и ss. Один шаблон должен задавать регистры ax, bx, cx, dx, ah, al, bh, bl, ch, cl, второй — bp и sp, третий — si и di, четвертый — ds, es, cs и ss.

Внимание: во всех шаблонах следует использовать квадратные скобки.

7. Выявить вещественные константы с фиксированной точкой в стиле языка Fortran 66. Незначащие нули вокруг точки необязательны, т. е. наряду с привычной формой записи (+0.125, -0.2 и -13.0) возможны сокращения (+.125, -.2 и 13.). Предусмотреть контроль переполнения.

Дополнение к заданию: в языке Fortran 66 сравнение записывается как .GT., .GE., .LT., .LE., .EQ., .NE. (вместо >, >=, <, <=, ==, <>), что приводит к таким конструкциям:

134.GT.0. — означает 134 > 0.0, т. к. первая точка относится не к числу 134, а к ".GT.";
+12..LT.-.1 — означает 12.0 < -0.1.

Как выяснить, какое число задано — целое или вещественное? Можно перечислить все эти сравнения в правилах. Есть более изящное решение с использованием концевого контекста.

8. Выводить в кавычках строковые константы, заданные в стиле языка Fortran. Имеется в виду строка в формате **nH**s, где n — число в диапазоне [1..255], которое задает длину многострочного текста s после буквы 'H'. Например, **5H**alfa234 означает "alfa2" (литеры 34

[15] Для преобразования строк в числа используйте C-функцию strtol.
[16] Если только в конце нет суффиксов q, xq, h, xh или xb.

уже не входят в строку). Текст, заданный таким образом, может распространяться на несколько строк, например:

```
30Habcdefghijklmnopqrstuvwxyz
123456789
```

задает строку, которую на C можно было бы определить так:

```
"abcdefghijklmnopqrstuvwxyz\n123"
```

Литеры 456789 уже не относятся к строковой константе. Они бы вошли в нее, если бы в начале было 36H.

9. Выявлять константы времени в стиле Step 7, например: T#1h30m25s123ms (без пробелов). Какие-то поля могут отсутствовать — как, например, в константах T#7s, T#3h456ms. Но порядок полей должен быть таким, как в примере.

Дополнительное задание: при помощи функции mktime проверить правильность данных. Эта функция принимает структуру с полями tm_hour, tm_min, tm_sec и т. п., а возвращает секунды (см., например, yacc/_date/v5/abs_date.c). Ошибку в данных можно обнаружить, сравнив значения структуры до и после вызова mktime. Например, 22:59:61 после mktime превратится в 23:00:01.

10. На входе задан ненумерованный библиографический список, по одному элементу в строке (см. texts/reference.txt). Получить список, отсортированный по году издания, при помощи Unix-утилиты **sort**; но прежде надо сделать две вспомогательные программы на lex:

- Первая (p1) находит в строке год издания (считаем для простоты, что первое попавшееся число от 1900 до нынешнего года и есть год издания) и помещает его в начало строки, отделив от названия каким-то значком — например, 1987@. Это вход утилиты **sort**.
- Вторая (p2) удаляет из каждой строки выходного потока утилиты **sort** начальное число со значком @.

Эти программы при вызове объединяются следующим образом: **p1 <1.in | sort | p2 >1.out**

11. На входе задан произвольный текст на английском языке. Получить список слов (исключив слишком короткие слова из 1–3 букв), отсортированный в алфавитном порядке и без повторов. Для этого воспользуйтесь утилитами **sort**[17] и **uniq**. Но прежде надо составить программу на lex, которая выявляет слова длиной > трех букв и выводит их по одному слову в строке. Выход этой программы подается на вход **sort** таким образом: **test <1.in | sort**

[17] В некоторых реализациях **sort** есть опция для исключения дублей; тогда **uniq** не нужна.

Приложение 2. Служебные литеры в регулярных выражениях Lex

В табл. П2.1 перечислены служебные литеры, используемые в шаблонах языка lex.

Таблица П2.1. Служебные литеры в шаблонах

Литера	Пример	Значение
"	"x"	**x**, даже если **x** — оператор
\	\x	
[ ]	[xy]	литера 'x' или 'y'
	[x-z]	литера в диапазоне от 'x' до 'z'
^	[^x]	любая литера кроме 'x'
	^x	**x** в начале строки
.	.	любая литера кроме конца строки
< >	<y>x	**x**, если стартовое состояние — **y**
$	x$	**x** в конце строки
?	x?	необязательное **x**
*	x*	0, 1, 2, ... экземпляров **x**
+	x+	1, 2, 3, ... экземпляров **x**
|	x|y	**x** или **y**
( )	(x)	**x**
/	x/y	**x**, но только если за ним **y**
{ }	{x}	макроподстановка **x**
	x{m}	m появлений **x**
	x{m,n}	от m до n появлений **x**
	x{m,}	m и более появлений **x**

Приложение 3. Контрольные вопросы по Lex

При ответе на вопросы, касающиеся шаблонов, составьте программу для проверки, взяв за основу пример ex4.l.

Таблица П3.1. Теоретические вопросы

1	Что означает {s} в шаблоне?
2	Что такое "пустое действие", как его задать? Что такое действие по умолчанию, когда оно выполняется?
3	Когда вызывается функция yywrap? Каким должно быть значение, возвращаемое этой функцией, чтобы работа сканера завершилась?
4	Где сохраняется распознанная входная последовательность? Как выяснить ее длину?
5	Какое действие задает макрокоманда ECHO?
6	Что произойдет с полученной входной последовательностью, если действие пустое?
7	Если входная последовательность не распознана ни одним правилом, что с ней будет?
8	Что хранится в переменных yytext и yyleng?

Таблица П3.2. Варианты практических задач

№	Каким должен быть шаблон для выявления:	
12	xyz xxzy xxyz xxyzzy xxxxxxxyzyzzyzyyzzy и т. д.	
	слова не более чем из 40 букв, где первая буква 'a', а в конце "yz"	
13	необязательного многоточия "…", за которым идут буквы и знак вопроса в конце	
	обозначений регистров i80386: eax, ebx, ecx, edx, esi, edi	
14	команд условных переходов ассемблера: jc, jnc, ja, jna, jb, jnb	
	пробелов и табуляций в конце строки	
15	ab abab ababab zab zabab zababab и т. д.	
	abc в начале строки, за которой идет пробел и любая из букв 'a', 'b' или 'c'	
16	гласных/согласных букв латинского алфавита	
	В чем разница между шаблонами? Продемонстрируйте на примерах.	
17	[^abc]	^abc
	a*	a+
18	a{1,}	a*
	[a|b]	a|b
19	a?b	[a?b]
	x$	x\n
20	x/\n	x\n
	.	“.”
21	Как можно задать, не повторяя, одинаковое действие для нескольких шаблонов? Запишите правила для копирования в выходной поток следующих последовательностей: John ab abab ababab zab zabab zababab. Должно быть два шаблона и одно, общее для них, действие	
22	Что произойдет, если действие для шаблона ‘John Smith’ содержит вызов yyless(3)? Что произойдет, если для этого же шаблона действие содержит вызов yymore(), в предположении, что прежде была распознана последовательность ‘USA, NY’?	
23	Как вернуть на вход распознанную последовательность или ее часть? Запрограммируйте действие, которое записывает обратно во входной поток 4 буквы — такие, чтобы при последующем вызове yylex() шаблон [a-z]+ выявил бы последовательность ‘when’	

Приложение 4. Десятичные коды ASCII для отладки программ на Lex/Yacc

Программы, написанные на lex и yacc, в отладочном режиме показывают принятые литералы в *десятичном* коде, и нужно понять, какие это литеры. Кодировка видимых литер приведена в табл. П5.1. Из служебных литер наиболее частые — 9 (табуляция), 10 (перевод строки) и 13 (возврат каретки).

Таблица П5.1. Литеры с кодами 32–127

Код	Литера	Код	Литера	Код	Литера	Код	Литера
32	пробел	56	8	80	P	104	h
33	!	57	9	81	Q	105	i
34	“	58	:	82	R	106	j
35	#	59	;	83	S	107	k
36	$	60	<	84	T	108	l
37	%	61	=	85	U	109	m
38	&	62	>	86	V	110	n
39	'	63	?	87	W	111	o
40	(	64	@	88	X	112	p
41	)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[	115	s
44	,	68	D	92	\	116	t
45	-	69	E	93	]	117	u
46	.	70	F	94	^	118	v
47	/	71	G	95	_	119	w
48	0	72	H	96	`	120	x
49	1	73	I	97	a	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	{
52	4	76	L	100	d	124	|
53	5	77	M	101	e	125	}
54	6	78	N	102	f	126	~
55	7	79	O	103	g	127	del

Приложение 5. Особенности работы в DOS/WinXP

Для работы в DOS/WinXP подходят трансляторы: Open Watcom Public License v1.0 или MS Visual C++ v5/6, flex v2.5 и byacc v1.9.

Вызов компилятора Watcom-C выполняется командой: **wcl *.c** (Watcom Compile & Link)[18]. В примерах из каталога works/c вместо функции fflush используйте flushall.

Запуск lex и yacc — по именам их exe-файлов, то есть: **flex** и **byacc**; ключи те же.

Сценарии **build.sh** переименуйте в **build.bat**, заменив в них **rm** на **del *.exe**, **lex** на **flex**, **yacc** на **byacc**, **cc** на **wcl**.

Перед началом работы преобразуйте тексты[19]: распакуйте архив works в Unix, перейдите в каталог works и вызовите сценарий tree.sh (в нем указана команда unix2dos); затем уберите комментарий в строке $cmd и повторите вызов[20].

В результате трансляции yacc-модуля вместо файлов y.tab.c и y.tab.h создаются y_tab.c и y_tab.h[21]; поэтому в lex-модулях нужно исправить директивы #include "y.tab.h".

Вместо файла y.output создается y.out.

Для включения настройки отладочного режима yacc:

- удалите определение переменной yydebug в файле zz.c;
- включите директиву "#define YYDEBUG 1" в секцию определений yacc-модуля;
- установите переменную окружения YYDEBUG равной единице (например, включив в autoexec.bat строку SET YYDEBUG=1).

[18] При работе с действительными числами рекомендуется указать ключ **/fpi87**.

[19] Конец строки в текстовых файлах DOS отмечается двумя кодами (cr и lf), а в Unix — только одним (lf). Некоторые (самые простые) редакторы в DOS и WinXP отображают такой текст в одну строку.

[20] Исходный вариант tree.sh выводит команды без выполнения.

[21] Причиной тому правила записи имен файлов в DOS: точка может быть одна, она отделяет имя файла (до 8 литер) от его расширения (до 3 литер).

Послесловие

В модулях, написанных на C (темы 1 и 2), имена глобальных функций и переменных совпадают с именами, принятыми в языках lex и yacc. Это позволяет использовать модули, написанные на C, с модулями на lex и yacc.

Дополнительные примеры по теме 3, в каталоге works/lex:

- В подкаталоге yywrap — пример программирования функции yywrap для включения во входной поток нескольких файлов[22]. Пробный вариант one_more.l в конце разбора делает переключение на файл “one_more.l” — один раз. Полнофункциональный вариант cmd_str.l последовательно переключает входной поток на все файлы, указанные в командной строке. Если в командной строке указан файл с перенаправлением, он обрабатывается, только если нет параметров[23]. Т. е. если задано **<test.in 1.in 2.in**, то файл test.in игнорируется, а на вход поступают 1.in и 2.in.

Дополнительные примеры по теме 4, в каталоге works/yacc:

- В каталоге name_table — разбор списка идентификаторов, с записью их в таблицу имен. Функции для работы с таблицей имен можно проверить отдельно: переименовать test.~c в test.c и скомпилировать программу из модулей test.c и nametab.c.
- В каталоге sa (structured assembler) — пример реализации управляющих конструкций структурного ассемблера. Структурный ассемблер — это ассемблер, в котором вместо команд переходов используются операторы языков высокого уровня: while-end, repeat-until, loop-end, if-else-elsif-end. Условия в while и until берутся из обозначений команд переходов. Например, команды jz и jnz, jc и jnc содержат условия z, nz, c и nc — и любое из них может использоваться справа от while, if, elsif и until. Задача программы — выявить структурные операторы и преобразовать их в команды ветвлений: {z, nz, c, nc} -> {jz, jnz, jc, jnc}. (Если условие выполнения нужно преобразовать в команду обхода, берется обратное условие: {z, nz} -> {jnz, jz}.) Если строка начинается не со слова while, repeat, end и т. д., то она считается оператором базового ассемблера и копируется в выходной поток без анализа (реализовано в lex-модуле при помощи стартовых условий). Результат трансляции — программа на языке базового ассемблера (в примере — на языке ассемблера для i80x86).
- В каталоге calc — калькулятор, он приведен в качестве примера yacc-спецификации для разбора арифметических выражений с заданием приоритета и ассоциативности [3].

Вопросы, не включенные в пособие:

- приоритетность и ассоциативность при разборе арифметических выражений [3];
- действия внутри правил [3];
- генерирование кода [4];
- средства, аналогичные lex и yacc, с выходом на языках C++ [5] и Java [6].

[22] Может пригодиться в системах, отличных от Unix. В Unix эта задача решается проще: **cat** *.in | ./a.out
[23] Файлы со значками перенаправления не считаются параметрами.