Теоретические сведения

Первая фаза компиляции называется **лексическим анализом** или сканированием.

Лексический анализатор (сканер) читает поток символов, составляющих исходную программу, и группирует эти символы в значащие последовательности, называющиеся *лексемами*.

Лексема – это структурная единица языка, которая состоит из элементарных символов языка и не содержит в своём составе других структурных единиц языка. Лексемами языков программирования являются идентификаторы, константы, ключевые слова языка, знаки операций и т.п.

На вход лексического анализатора поступает текст исходной программы, а выходная информация передаётся для дальнейшей обработки синтаксическому анализатору. Для каждой лексемы сканер строит выходной *токен* (англ. token – знак, символ) вида

‹имя_токена, значение_атрибута›

Первый компонент токена, *имя_токена*, представляет собой абстрактный символ, использующийся во время синтаксического анализа, а второй компонент, значение атрибута, указывает на запись в таблице идентификаторов, соответствующую данному токену.

Предположим, например, что исходная программа содержит инструкцию присваивания

$$a = b + c * d$$

Символы в этом присваивании могут быть сгруппированы в следующие лексемы и отображены в следующие токены:

1) *a* представляет собой лексему, которая может отображаться в токен ‹*id,* 1›, где *id* – абстрактный символ, обозначающий идентификатор, а 1 указывает запись в таблице идентификаторов для a, в которой хранится такая информация как имя и тип идентификатора;

2) символ присваивания = представляет собой лексему, которая отображается в токен ‹=›. Поскольку этот токен не требует значения атрибута, второй компонент данного токена опущен. В качестве имени токена может быть использован любой абстрактный символ, например, такой, как «assign», но для удобства записи в качестве имени абстрактного символа можно использовать саму лексему;

3) *b* представляет собой лексему, которая отображается в токен ‹*id,* 2›, где 2 указывает на запись в таблице идентификаторов для b;

4) + является лексемой, отображаемой в токен ‹+›;

5) *c* – лексема, отображаемая в токен ‹*id*, 3›, где 3 указывает на запись в таблице идентификаторов для c;

6) * – лексема, отображаемая в токен ‹*›;

7) *d* – лексема, отображаемая в токен ‹*id*, 4›, где 4 указывает на запись в таблице идентификаторов для d.

Пробелы, разделяющие лексемы, лексическим анализатором пропускаются.

Представление инструкции присваивания после лексического анализа в виде последовательности токенов примет следующий вид:

‹*id*, 1›‹=›‹*id*, 2›‹+›‹*›‹*id*, 3›‹*id*, 4›.

С теоретической точки зрения лексический анализатор не является обязательной, необходимой частью компилятора. Его функции могут выполняться на этапе синтаксического разбора. Однако существует несколько причин, исходя из которых в состав практически всех компиляторов включают лексический анализ:

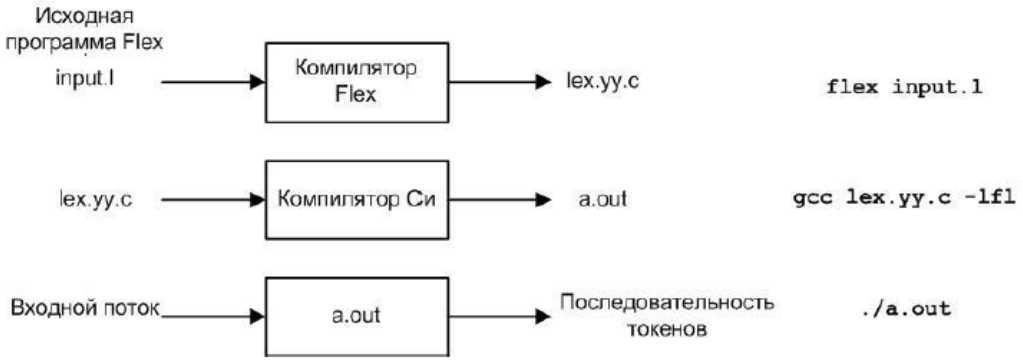
- упрощается работа с текстом исходной программы на этапе синтаксического разбора и сокращается объём обрабатываемой информации, так как лексический анализатор
- структурирует поступающий на вход исходный текст программы и удаляет всю незначащую информацию;
- для выделения в тексте и разбора лексем можно применять простую, эффективную и теоретически хорошо проработанную технику анализа, в то время как на этапе синтаксического анализа конструкций исходного языка используются достаточно сложные алгоритмы разбора;
- сканер отделяет сложный по конструкции синтаксический анализатор от работы непосредственно с текстом исходной программы, структура которого может варьироваться в зависимости от версии входного языка – при такой конструкции компилятора при переходе от одной версии языка к другой достаточно только перестроить относительно простой сканер.

Функции, выполняемые лексическим анализатором, и состав лексем, которые он выделяет в тексте исходной программы, могут меняться в зависимости от версии компилятора. В основном лексические анализаторы выполняют исключение из текста исходной программы комментариев и незначащих пробелов, а также выделение лексем следующих типов: идентификаторов, строковых, символьных и числовых констант, ключевых (служебных) слов входного языка.

Генератор лексических анализаторов Flex

Существуют различные программные средства для решения задачи построения лексических анализаторов. Наиболее известным из них является Lex (в более поздних версиях – Flex).

Программный инструментарий Flex позволяет определить лексический анализатор с помощью регулярных выражений для описания шаблонов токенов. Входные обозначения для Flex обычно называют *языком Flex*, а сам инструмент – *компилятором Flex*. Компилятор Flex преобразует входные шаблоны в конечный автомат и генерирует код (в файле с именем lex.yy.c), имитирующий данный автомат.



Входной файл input.l написан на языке Flex и описывает генерируемый лексический анализатор. Компилятор Flex преобразует `input.l` в программу на языке программирования Си (файл с именем `lex.yy.c`). При компиляции lex.yy.c необходимо прилинковать библиотеку Flex (`-lfl`). Этот файл компилируется в файл с именем `a.out` как обычно. Выход компилятора Си представляет собой работающий лексический анализатор, который на основе потока входных символов выдаёт поток токенов.

Обычно полученный лексический анализатор, используется в качестве подпрограммы синтаксического анализатора.

Структура программы на языке Flex имеет следующий вид:

```
Объявления
%%
Правила трансляции
%%
Вспомогательные функции
```

Обязательным является наличие правил трансляции, а, следовательно, и символов %% перед ними. Правила могут и отсутствовать в файле, но %% должны присутствовать всё равно.

Раздел объявлений может включать объявления переменных, именованные константы и регулярные определения (например, `digit [0-9]` – регулярное выражение, описывающее множество цифр от 0 до 9). Кроме того, в разделе объявлений может помещаться символьный блок, содержащий определения на Си. Символьный блок всегда начинается с `%{` и заканчивается `%}`. Весь код символьного блока полностью копируется в начало генерируемого файла исходного кода лексического анализатора.

Второй раздел содержит правила трансляции вида

```
Шаблон { Действие }
```

Каждый шаблон является регулярным выражением, которое может использовать регулярные определения из раздела объявлений. Действия представляют собой фрагменты кода, обычно написанные на языке программирования Си, хотя существуют и разновидности Flex для других языков программирования.

Третий раздел содержит различные дополнительные функции на Си, используемые в действиях. Flex копирует эту часть кода в конец генерируемого файла.

Специальные символы, использующиеся в регулярных выражениях Flex

Символ шаблона	Значение
.	Соответствует любому символу, кроме `\n`
[]	Класс символов, соответствующий любому из символов, описанных внутри скобок. Знак `'-'` указывает на диапазон символов. Например, `[0-9]` означает то же самое, что и `[0123456789]`, `[a-z]` – любая прописная буква латинского алфавита, `[A-z]` – все заглавные и прописные буквы латинского алфавита, а также 6 знаков пунктуации, находящихся между `Z` и `a` в таблице ASCII. Если символ `'-'` или `']'` указан в качестве первого символа после открывающейся квадратной скобки, значит он включается в описываемый класс символов. Управляющие (escape) последовательности языка Си также могут указываться внутри квадратных скобок, например, `\t`.
^	Внутри квадратных скобок используется как отрицание, например, регулярное выражение `[^\t\n]` соответствует любой последовательности символов, не содержащей табуляций и переводов строки. Если просто используется в начале шаблона, то означает начало строки.
$	При использовании в конце регулярного выражения означает конец строки.
{}	Если в фигурных скобках указаны два числа, то они интерпретируются как минимальное и максимальное количество повторений шаблона, предшествующего скобкам. Например, `A{1,3}`

Символ шаблона	Значение
	соответствует повторению буквы A от одного до трёх раз, а `0{5}` – `00000`. Если внутри скобок находится имя регулярного определения, то это просто обращение к данному определению по его имени.
\	Используется в escape-последовательностях языка Си и для задания метасимволов, например, `\*` – символ '*' в отличие от * (см. ниже).
*	Повторение регулярного выражения, указанного до *, 0 или более раз. Например, `[ \t]*` соответствует регулярному выражению для пробелов и/или табуляций, отсутствующих или повторяющихся несколько раз.
+	Повторение регулярного выражения, указанного до +, один или более раз. Например, `[0-9]+` соответствует строкам 1, 111 или 123456.
?	Соответствует повторению регулярного выражения, указанного до ?, 0 или 1 раз. Например, `-?[0-9]+` соответствует знаковым числам с необязательным минусом перед числом.
|	Оператор «или». Например, `true|false` соответствует любой из двух строк.
()	Используются для группировки нескольких регулярных выражений в одно. Например, `a(bc|de)` соответствует входным последовательностям: abc или ade.
/	Так называемый присоединенный контекст. Например, регулярное выражение `0/1` соответствует 0 во входной строке 01, но не соответствует ничему в строках 0 или 02.
" "	Любое символы в кавычках рассматриваются как строка символов. Метасимволы, такие как *, теряют своё значение и интерпретируются как два символа: \ и *.

Встроенные переменные

- `yytext[]` – одномерный массив (последовательность символов), содержащий фрагмент входного текста, удовлетворяющего регулярному выражению и распознанного данным правилом;
- `yyleng` – целая переменная, значение которой равно количеству символов, помещенных в массив `yytext`.

Встроенные переменные позволяют определить конкретную последовательность символов, распознанных данным правилом. При применении правила анализатор lex автоматически заполняет значениями встроенные переменные. Эти значения можно использовать в действии примененного правила.

Пример правила:

```
[a-z]+ printf("%s",yytext);
```

Регулярное выражение правила определяет бесконечное множество последовательностей символов, состоящих из букв латинского алфавита. Данное правило применяется, когда из входного потока символов поступает конкретная последовательность символов, удовлетворяющих его регулярному

выражению. Оператор языка C printf выводит в выходной поток эту последовательность символов.

Встроенные функции

`yymore()`

– В обычной ситуации содержимое `yytext` обновляется всякий раз, когда производится применение некоторого правила. Иногда возникает необходимость добавить к текущему содержимому `yytext` цепочку символов, распознанных следующим правилом. `yymore()` вызывает переход анализатора к применению следующего правила. Входная последовательность символов, распознанная следующим правилом, будет добавлена в массив `yytext`, а значение переменной `yyleng` будет равно суммарному количеству символов, распознанными этими правилами.

`yyless(n)`

– Оставляет в массиве `yytext` первые n символов, а остальные возвращает во входной поток. Переменная `yyleng` принимает значение n. Лексический анализатор будет читать возвращенные символы для распознавания следующей лексемы. Использование `yyless(n)` позволяет посмотреть правый контекст.

`input()`

– Выбирает из входного потока очередной символ и возвращает его в качестве своего значения. Возвращает ноль при обнаружении конца входного потока.

`output(c)`

– Записывает символ c в выходной поток.

`unput(c)`

– Помещает символ c во входной поток.

`yywrap()`

– Автоматически вызывается при обнаружении конца входного потока. Если возвращает значение 1, то лексический анализатор завершает свою работу, если 0 – входной поток продолжается текстом нового файла. По умолчанию `yywrap` возвращает 1. Если имеется необходимость продолжить ввод данных из другого источника, пользователь должен написать свою версию функции `yywrap()`, которая организует новый входной поток и возвратит 0.

Примеры

1. Анализатор идентификатора

```
%option noyywrap
%{
     #include<stdio.h>
     #include<locale.h>
%}
letter [a-z]
digit [0-9]
identifier {letter}({letter}|{digit})*
%%
{identifier} {printf("идентификатор %s\n",yytext);}
%%
int main()
{
     setlocale(LC_ALL, "Rus");
     printf("Enter a string\n");
     yylex();
     //return 0;
}
```

Lexer принимает на входе поток символов, а когда встречает группу символов, совпадающих с некоторым шаблоном (ключом), выполняет определенное действие.

В первой секции между парой скобок %{ и %} мы включили текст, который напрямую попадет в итоговую программу. Это включение необходимо, потому что впоследствии мы используем подпрограмму printf, которая определена в заголовочном файле stdio.h и возможность локализации русского языка (locale.h).

Затем описываются нетерминалы (в Lex называются определениями) – letter и digit. Затем определяется идентификатор (фигурные скобки применяются для обособления уже определенных величин).

В секции Правил описываем правила трансляции лексера – при каждом распознавании идентификатора выполняется простой оператор вывода.

В функции main вызывается yylex() – функция, непосредственно выполняющая лексический анализ входного текста.

2. Анализатор действительных чисел

```
%option noyywrap
%{
     #include<stdio.h>
     #include<locale.h>
%}
digit [0-9]
realno [+\-]?{digit}*\.{digit}+

%%
{realno}        {printf("действительное        число        %s
опознано\n",yytext);}
%%
int main()
{
     setlocale(LC_ALL, "Rus");
     printf("Enter a string\n");
     yylex();
     //return 0;
}
```

a	представляет отдельный знак;
\a	представляет a, если a — знак, используемый в системе обозначений (для устранения неоднозначности);
"a"	также представляет a, если a — знак, используемый в системе обозначений;
a|b	представляет a или b;
a?	представляет нуль или одно вхождение a;
a*	представляет нуль или более вхождений a;
a+	представляет одно или более вхождений a;
a{m,n}	представляет от m до n вхождений a;
[a-z]	представляет набор знаков (алфавит);
[a-zA-Z]	также представляет набор знаков (больший);
[^a-z]	представляет дополнение первого набора знаков;
{name}	представляет регулярное выражение, определенное идентификатором name;
^a	представляет a в начале строки;
a$	представляет a в конце строки;
ab\xy	представляет ab, следующее перед xy.

3. Анализатор текста

Допустим, нам нужно проанализировать файл следующего вида:

```
logging {
        category lame-servers { null; };
        category cname { null; };
};

zone "." {
        type hint;
        file "/etc/bind/db.root";
};
```

Ясно видим, что здесь встречаются следующие категории (токены):

- Слова (WORD), например 'zone' и 'type'
- Имена файлов (FILENAME), например '/etc/bind/db.root'
- Кавычки (QUOTE), к примеру, те, которые окружают имя файла
- Открывающие фигурные скобки (OBRACE) - {
- Закрывающие фигурные скобки (EBRACEs) - }
- Точка с запятой (SEMICOLON) - ;

Lex-файл:

```
%option noyywrap
%{
     #include<stdio.h>
     #include<stdlib.h>
     #include <locale.h>
%}

%%
[a-zA-Z][a-zA-Z0-9]*    printf("WORD ");
[a-zA-Z0-9\/.-]+        printf("FILENAME ");
\"                      printf("QUOTE ");
\{                      printf("OBRACE ");
\}                      printf("EBRACE ");
;                       printf("SEMICOLON ");
\n                      printf("\n");
[ \t]+           /* игнорируем пробелы и знаки табуляции */;
%%

int main()
{
setlocale(LC_ALL, "Rus");
yylex();
}
```

4. Анализатор языка, содержащий операторы цикла for (…; …; …) do …, разделённые символом ; (точка с запятой). Операторы цикла содержат идентификаторы, знаки сравнения <, >, =, десятичные числа с плавающей точкой (в обычной и экспоненциальной форме), знак присваивания (:=).

```
%option noyywrap yylineno
%{
     #include <stdio.h>
     int ch;
%}
digit[0-9]
letter[a-zA-Z]
delim[();]
oper[<>=]
ws[ \t\n]
%%
for { printf("KEYWORD (%d, %d): %s\n", yylineno, ch, yytext);
ch += yyleng;
}

do { printf("KEYWORD (%d, %d): %s\n", yylineno, ch, yytext);
ch += yyleng;
}

("_"|{letter})("_"|{letter}|{digit})* {
printf("IDENTIFIER (%d, %d): %s\n", yylineno, ch, yytext);
ch += yyleng;
}

[-+]?({digit}*\.{digit}+|{digit}+\.|{digit}+)        ([eE][-
+]?{digit}+)?[flFL]? {
printf("NUMBER (%d, %d): %s\n", yylineno, ch, yytext);
ch += yyleng;
}

{oper} { printf("OPERATION (%d, %d): %s\n", yylineno, ch,
yytext);
ch += yyleng;
}

":=" { printf("OPERATION (%d, %d): %s\n", yylineno, ch,
yytext);
ch += yyleng;
}

{delim} { printf("DELIMITER (%d, %d): %s\n", yylineno, ch,
yytext);
```

```
     ch += yyleng;
     }

     {ws}+ { ch += yyleng; }

     . { printf("Unknown character (%d, %d): %s\n", yylineno, ch,
yytext);
     ch += yyleng;
     }
     %%
     int main(int argc, char **argv)
     {
          if(argc < 2)
          {
               printf("\nNot  enough  arguments.  Please  specify
          filename.\n");
               return -1;
          }
          if((yyin = fopen(argv[1], "r")) == NULL)
          {
               printf("\nCannot open file %s.\n", argv[1]);
               return -1;
          }
          ch = 1;
          yylineno = 1;
          yylex();
          fclose(yyin);
          return 0;
     }
```

(В коде есть ошибка, найдите ее и исправьте)

В современных версиях Flex рекомендуется отключать использование yywrap с помощью опции noyywrap, если в программе на языке Flex есть своя функция main, в которой и определяется, какой файл и когда сканировать.

Использование опции yylineno позволяет вести нумерацию строк входного файла и в случае ошибки сообщать пользователю номер строки, в которой эта ошибка произошла. Flex определяет переменную yylineno и автоматически увеличивает её значение на 1, когда встречается символ '\n'. При этом Flex не инициализирует эту переменную. Поэтому в функции main перед вызовом функции лексического анализа yylex переменной yylineno присваивается 1.

Flex, по умолчанию, присваивает переменной yyin указатель на стандартный поток ввода. Если предполагается сканировать текст из файла, то нужно присвоить переменной yyin результат вызова функции fopen до вызова yylex:

yyin = fopen(argv[1], "r");

В функции main в приведённом примере открывается файл, имя которого было указано пользователем при вызове лексического анализатора.

Входной текстовый файл prog содержит следующую программный код:

for(abc1:=.;abc1<11.0E+1;abc1:=abc1>.1)do abc1:=abc1=34E5;

Для компиляции и запуска программы используются следующие команды:

flex example.l

gcc lex.yy.c -o scanner –lfl

./scanner prog