5.4 Match-Algorithmus

Der **generierte Scanner sucht** in seiner Eingabe nach **Strings**, auf die eines der **Muster paßt**.

Falls **mehrere passen**, nimmt er die Regel, auf die **am meisten Text** paßt.

Falls zwei Matches die **gleiche Länge** haben, nimmt er die Regel, die **zuerst** in der Lex-Datei steht.

Wenn der Match feststeht, wird der Text des Tokens in der **Variablen** `yytext` bereitgestellt, und die Länge in `yyleng`.

Dann wird die zugehörige **Aktion ausgeführt**.

Dann geht es wieder **von vorne** los.

Falls **kein Match** gefunden wurde, gilt die **Default-Regel**: Ein einzelnes Zeichen paßt und wird gedruckt.

Das einfachste `flex`-Programm:
`trivial.l`:

```
%option main
%%
```

▶▶ `make trivial`

▶▶ `./trivial < pcount.txt`

Durch die **Option** `main` wird eine einfache **main()-Funktion generiert**, die die Scanner-Funktion yylex() aufruft.

Am **Ende der Eingabe** kehrt **yylex()** mit dem **Return-Wert 0** zurück.

Man kann auch **innerhalb einer Aktion return(n)** ausführen lassen. Wird **yylex()** danach **erneut aufgerufen**, macht sie an der **nächsten Position** einfach weiter.

Den **Return-Wert** kann man z.B. **verwenden,** um die **Art des** gefundenen **Tokens** an den Aufrufer mitzuteilen. Auch die **globalen Variablen** `yytext` und `yyleng` **bleiben** für den Aufrufer erhalten.

5.5 Lesen aus Strings

Wir wollen die **Kommandozeile** für ein einfaches Programm **auswerten**. (Wir wollen dabei **nicht** die **getopt()-Funktion** aus der C-Bibliothek nehmen, sondern selbst eine schreiben.)

Man kann auch **aus Strings** statt aus Dateien **lesen**.

Leider geht das bei verschiedenen **Versionen von Lex** mit **verschiedener Syntax**.

Bei Flex schaltet man mit der **Funktion** yy_scan_string() die Eingabe um:

`cmdline.l`:

```
%{
#include <stdio.h>
int verbose = 0;
char *progName = NULL;
%}
%option noyywrap
%%
^"-h"           |
^"-?"           |
^"--help"       { printf("usage is: %s [--help | -h | -?] ", progName);
                  printf("[--verbose | -v ...]\n");
                  exit(0);
                }
^"-v"           |
^"--verbose"    { verbose++; }
.*              { printf("unknown option '%s'!\n", yytext);
                  exit(1);
                }
%%
int main(int argc, char *argv[]) {
    progName = *argv;
    while(++argv,--argc) {
        yy_scan_string(*argv);
        yylex();
    }
    printf("Now starting to frobnicate with:\n");
    printf("verbose = %d\n", verbose);
}
```

5.6 Startbedingungen

Nun möchten wir eine **Option mit Parameter** dazufügen. Sie soll einen **Dateinamen** angeben:

cmdline2.l:

```
%{
#include <stdio.h>
#include <string.h>
int verbose = 0;
char *progName = NULL;
char *filename = "-";
%}
%option noyywrap
%x FNAME
%%
^"-h"           |
^"-?"           |
^"--help"       { printf("usage is: %s [--help | -h | -?] ", progName);
                  printf("[--verbose | -v ...] ");
                  printf("[(--file | -f) filename]\n");
                  exit(0);
                }
^"-v"           |
^"--verbose"    { verbose++; }
^"-f"           |
^"--file"       { BEGIN FNAME;
                  filename = "";
                }
.*              { printf("unknown option '%s'!\n", yytext);
                  exit(1);
                }
<FNAME>.+       { filename = strdup(yytext); BEGIN INITIAL; }
%%
int main(int argc, char *argv[]) {
    progName = *argv;
    while(++argv,--argc) {
        yy_scan_string(*argv);
        yylex();
    }
    if(!filename || !*filename) {
        printf("No filename given with option --file!\n");
        exit(1);
    }
    printf("Now starting to frobnicate with:\n");
    printf("verbose = %d\n", verbose);
    printf("file = '%s'\n", filename);
}
```

Nach der Option `-f` muß **als nächstes** der **Dateiname** kommen. Es kann **keine weitere Option** dort stehen. Die **normalen Regeln** müssen **abgeschaltet** werden, und eine **besondere Regel** für den Dateinamen muß **eingeschaltet** werden.

Dies erreichen wir mit einer **Startbedingung**. Eine Startbedingung benutzt eine **zusätzliche Zustandsvariable** des Scanners.

Oben definieren wir die Startbedingung mit `%x`.

Nach Ausführung des **Makros** `BEGIN FNAME` sind **nur noch Regeln aktiv**, vor denen in **spitzen Klammern** der Name der Startbedingung steht.

Mit `BEGIN INITIAL` kommt man in den **initialen Zustand** zurück.

Man kann mit `%x` **beliebig viele** solche **Zustände** definieren. **Intern** handelt es sich um eine **Integer-Variable**, die gesetzt und abgeprüft wird. Z.B. ist `INITIAL` **immer als 0** definiert.

Innerhalb der Regeln kann man den **Zustand** über das **Makro** `YY_START` **lesen** und z.B. einer Integer-Variablen zuweisen. (Bei **Flex** gibt es auch ein Makro `YYSTATE` mit der gleichen Wirkung.)

Man beachte im Beispiel, wie in `main()` **geprüft** wird, **ob** nach `-f` wirklich noch ein **Dateiname angegeben** wurde. (`YYSTATE` dürfen wir **außerhalb** der Regeln leider **nicht** verwenden.)

Soll eine **Regel in mehr als einem Zustand aktiv** sein, kann man die Startbedingungen mit **Komma** getrennt in den spitzen Klammern aufführen.

Gibt man einen **Stern** `*` als Startbedingung an, gilt die Regel in **allen Zuständen**.

5.7 Online-Aufgabe: C-Quellcode-Zähler

Schreibe ein Programm, daß eine **C-Quell-Datei** einliest und **zählt**, wieviele Zeilen es jeweils hat mit:

- C-Code
- nur Kommentar
- nur Whitespace

Hinweis zum **Algorithmus**: Lese die Eingabe in kleinen Stücken ein. Benutze Flags, um dir zu merken, ob du in dieser Zeile schon C-Code bzw. einen Kommentar gesehen hast. Wenn Du ein Newline siehst, dann zähle die C-Code-, Kommentar- und Whitespace-Zeilenzähler entsprechend weiter und setze die Flags zurück.

Hinweis zum Programmaufbau: Die Verwendung von **Startbedingungen** ist **hilfreich**.

Eine **Beispiel-Eingabe-Datei**:
`cexample.c`:

```
/*
Dies ist eine Test-Datei fuer den C-Code-Zaehler.
*/

int main() {
    int idx; /* Index */
    int count; /* Dies ist
                  ein Zaehler */

    /* Buchstabe:
    */ char letter;
}
```

Zusatzaufgabe: Es soll auch der Fall richtig gehandhabt werden, daß ein **String** einen **Kommentar-Anfang oder -Ende enthält**. Strings dürfen auch über **mehrere Zeilen** gehen, und sie dürfen `\"` enthalten.

Eine **Beispiel-Eingabe-Datei**:
`cexample2.c`:

```
/*
Dies ist eine Test-Datei fuer den erweiterten C-Code-Zaehler.
*/

int main() {
    int idx; /* Index */
    int count; /* Dies ist
                  ein Zaehler */

    /* Buchstaben:
    */ char *letters = "Hello\
\
    world\n";
}
```

5.8 Mehrere Eingabequellen nacheinander

Beim Scannen der **Kommandozeile** auf Seite 157 haben wir **alle Parameter separat** durch Aufrufe von `yylex()` gelesen. **Nachteil** war, daß `yylex()` immer mit `return` **zurückkehrte**.

Es geht auch **anders**: Wenn `yylex()` **am Ende** eines Strings bzw. einer Datei angekommen ist, ruft es die **Funktion** `yywrap()` auf. Falls `yywrap()` den Wert **false (d.h. 0)** zurückliefert, dann **macht** `yylex()` **einfach weiter** und nimmt an, daß die **Eingabe** nun auf **neue Daten zeigt**.

Wenn man **keine eigene Funktion** `yywrap()` schreibt, dann muß man die **Option** `%noyywrap` angeben, wie wir es bisher immer getan haben.

Beispiel:
`cmdline-wrap.l`:

```
%{
#include <stdio.h>
int verbose = 0;
char *progName = NULL;
char **currArg = NULL;
int currArgNum = 0;
%}
%%
^"-h"           |
^"-?"           |
^"--help"       { printf("usage is: %s [--help | -h | -?] ", progName);
                  printf("[--verbose | -v ...]\n");
                  exit(0);
                }
^"-v"           |
^"--verbose"    { verbose++; }
.*              { printf("unknown option '%s'!\n", yytext);
                  exit(1);
                }
%%

int yywrap() {
    if(--currArgNum <= 0)
        return 1;
    yy_scan_string(*(++currArg));
    return 0;
}

int main(int argc, char *argv[]) {
    progName = *argv;
    currArg = ++argv;
    currArgNum = --argc;
    if(currArgNum > 0) {
        yy_scan_string(*currArg);
        yylex(); /* Nur EIN Aufruf fuer alle Parameter */
    }
    printf("Now starting to frobnicate with:\n");
    printf("verbose = %d\n", verbose);
}
```

Das gleiche geht natürlich **auch** für **mehrere Dateien** nacheinander. Dann muß man `yyin` **neu zuweisen**. So eine Zuweisung haben wir **oben** schon **gesehen**, als wir die **Pascal-artige** Sprache aus einer (einzigen) **Datei gelesen** haben.

5.9 Mehrere Eingabequellen abwechselnd

Wenn man eine Eingabesprache scannt, die **„include"-Anweisungen** enthält, muß man **mitten im Datenstrom** auf eine andere Eingabe **umschalten** und später **zurückschalten**. Hier kann man z.B. `yyin` **nicht** einfach **neu zuweisen**, weil der Lexer oft ein großes Stück **vorausliest und puffert**. Erst viele Zeilen **später** würde die Zuweisung **wirksam werden**.

Daher kann man **mehrere Eingabepuffer** haben und zwischen ihnen **hin- und herschalten**.

Beispiel:
`include.l`:

```
%{
#include <stdio.h>
#define MAX_INCLUDE_DEPTH 10
YY_BUFFER_STATE include_stack[MAX_INCLUDE_DEPTH];
int include_stack_pos = 0;
%}
%option main
%x INCL
%%
^"#include"[ \t]+   BEGIN(INCL);
.|\n                ECHO;
<INCL>.*\n          { if(include_stack_pos >= MAX_INCLUDE_DEPTH) {
                        fprintf(stderr,"Includes nested too deeply!\n");
                        exit(1);
                      }
                      include_stack[include_stack_pos++] =
                        YY_CURRENT_BUFFER;
                      yytext[strlen(yytext)-1] = '\0'; /* remove '\n' */
                      if(!(yyin = fopen(yytext, "r"))) {
                        fprintf(stderr,
                          "Cannot open include file '%s'!\n", yytext);
                        exit(1);
                      }
                      yy_switch_to_buffer(
                        yy_create_buffer(yyin, YY_BUF_SIZE));
                    BEGIN(INITIAL);
                  }
<<EOF>>             { if(--include_stack_pos < 0)
                        yyterminate();
                      else {
                        yy_delete_buffer(YY_CURRENT_BUFFER);
                        yy_switch_to_buffer(
                          include_stack[include_stack_pos]);
                      }
                  }
```

Der Zustand **INCL** dient dazu, den Dateinamen einzulesen.

Dabei ist wichtig, das **Newline** zu lesen, bevor das Include ausgeführt wird. Allerdings muß es vom Dateinamen abgetrennt werden.

Das Makro `ECHO` druckt das aktuelle Token aus `yytext` einfach zur Ausgabedatei `yyout` aus.

Analog zur Variablen `yyin`, die auf die Eingabedatei zeigt, zeigt die **Variable** `yyout` auf die **Ausgabedatei** (Default `stdout`). Sie kann ebenfalls **neu zugewiesen** werden.

YY_BUFFER_STATE ist ein opaquer Datentyp, der genau einen Zeiger auf einen Eingabepuffer faßt.

Die Variable **YY_CURRENT_BUFFER** enthält den Zeiger auf den aktuellen Eingabepuffer.

Mit **yy_switch_to_buffer(...)** kann man auf einen anderen Eingabepuffer umschalten. Man kann es auch sonst anstelle eine Zuweisung auf `yyin` verwenden.

Mit **yy_create_buffer(yyin, YY_BUF_SIZE)** kann man einen neuen Eingabepuffer erzeugen.

MIt **yy_delete_buffer(...)** kann man einen Eingabepuffer wieder freigeben.

yyterminate() führt im wesentlichen ein `return 0;` aus.

Außerdem gibt es noch **yy_flush_buffer(YY_BUFFER_STATE buffer)**, um einen Eingabepuffer zu leeren und wie neu zu machen.

5.10 Aufruf- und Datei-Optionen von Flex

Man kann Flex eine Reihe von **Kommandozeilen-Optionen** mitgeben. Alternativ kann man fast alle auch **in** der **Datei** per `%o...` angeben.

Die wichtigsten Aufruf- und Datei-Optionen:

Aufruf- und Datei-Optionen von Flex

Datei	Aufruf	Bedeutung
`%option main`		Generiere eine `main()`-Funktion. Impliziert `noyywrap`.
`%option noyywrap`		Generiere eine `yywrap()`-Funktion.
`%option stdout`	`-t`	Der generierte Scanner geht nach `stdout` statt nach `lex.yy.c`.
`%option case-insensitive`	`-i`	Beim Matchen werden Groß- und Kleinbuchstaben nicht unterschieden.
`%option nodefault`	`-s`	Generiere keine Default-Regel. Falls keine Regel paßt, gibt es eine Fehlermeldung.
	`--help`	Gibt die möglichen Aufrufoptionen aus.
	`--version`	Gibt die Versionsnummer aus.
`%option yylineno`		Die Variable `yylineno` enthält immer die aktuelle Zeilennummer.

Kurs Unix-Tools, WS 01/02, Jan Bredereke 41

Weitere Aufruf- und Datei-Optionen von Flex

Datei	Aufruf	Bedeutung
`%option lex-compat`	`-l`	Maximale Kompatibilität mit originalem `lex` von AT&T.
`%option c++`	`-+`	Generiere eine C++-Scanner-Klasse.
`%option debug`	`-d`	Der generierte Scanner macht Debug-Ausgaben, wenn die Variable `yy_flex_debug` ungleich Null.
`%option perf-report`	`-p`	Gibt Performance-Report auf `stdout` aus, der „teure“ benutzte Features auflistet.
`%option prefix="`*XYZ*`"`	`-P`*XYZ*	Ersetzt in allen Variablen- und Funktionsnamen das Präfix yy durch das angegebene Präfix.
`%option stack`		Erlaubt die Benutzung von Stacks von Startbedingungen.
...	...	...

Kurs Unix-Tools, WS 01/02, Jan Bredereke 42

5.11 Weitere Features

Flex kennt eine ganze Reihe von **weiteren Features**. **Einige** davon werden hier noch kurz vorgestellt. **Ansonsten** siehe das **Manual**.

Man kann beim **Matchen** von Mustern auf die **nächsten Zeichen vorausschauen**, **ohne** daß sie **zum gematchten Text gehören** werden:

```
"include "/\"[^"\n]*\"
```

matcht nur dann den Text „include “, wenn auch ein Dateiname folgt.

Eine bereits **bekannte** Art von **Zeichen-Vorschau** ist der **reguläre Ausdruck** `$`, der auf ein folgendes Zeilenende prüft.

Eine **weitere** Art von Vorschau erreicht man mit der **Funktion** `yyless()`. Mit ihr kann man innerhalb einer Aktion den **letzten Teil der gelesenen Buchstaben wieder zurück** in die Eingabe schieben. Der **Unterschied** zur Vorschau mit `/` ist, daß `yytext` zwischenzeitlich den gesamten Text enthält, und daß `yyleng` entsprechend die gesamte Länge enthält. Der **Parameter** von `yyless()` ist die Anzahl der *nicht* mehr zurückzuschiebenden Zeichen.

Das **Makro** `REJECT` dient ähnlichen Zwecken. Es **bricht** die augenblickliche **Aktion ab** und **springt zur** am **nächstbesten passenden Regel**. Ein **Beispiel** wäre ein **Wort-Zähl-Programm**, daß zusätzlich bei einem bestimmten **Schlüsselwort** eine Funktion ausführt, aber danach trotzdem noch das Schlüsselwort als Wort mitzählt.

In regulären Ausdrücken kann man wie bekannt **Buchstabenmengen** verwenden. **Außer Buchstabenbereichen** kann man **auch** die folgenden Ausdrücke verwenden:

```
[:alnum:] [:alpha:] [:digit:]
[:blank:] [:space:]
[:lower:] [:upper:]
...
```

Verwendungsbeispiel:

```
[[:alpha:]0-9]+
```

(ist äquivalent zu `[[:alnum:]]+`)

Die Mengen werden **definiert über** die Funktionen der **C-Bibliothek** `isalnum()`, ...

Diese Buchstabenklassen sind eine **Flex-Erweiterung** von Lex.

Normalerweise haben wir **Startbedingungen** mit

```
%x FOO
```

definiert. Wenn wir mit `BEGIN(FOO)` in diesen Startzustand gewechselt sind, dann waren *nur* noch exklusiv die Regeln **dieses Startzustandes** aktiv.

Wenn man eine Startbedingung mit

```
%s FOO
```

definiert, dann haben wir eine **inklusive Startbedingung**, und es sind im entsprechenden Zustand außerdem immer noch alle Regeln aktiv, die **keinen Startbedingung** haben.

Man kann damit **gut Default-Regeln** schreiben, aber man muß gut **aufpassen**, daß nicht zuviele Regeln aktiv bleiben. **Default-Regeln** kann man **auch mit** der **Startbedingung** `<*>` schreiben. Leider kennt die **originale Version von Lex nur inklusive** Startbedingungen.

Wenn man einen ganzen Satz von **Regeln** hat, die **alle zur selben Startbedingung** gehören, dann muß man die Startbedingung **nicht vor jede** der Regeln schreiben.

```
<STRING>foo     ...
<STRING>bar     ...
```

Stattdessen kann man auch einen **Startbedingungsbereich** definieren:

```
<STRING>{
foo    ...
bar    ...
}
```

Manchmal **reicht** der **endliche Automat** des **Startzustandes nicht** aus. Dann kann man bei Flex einen **Stack von Startbedingungen** benutzen. Dafür gibt es

```
void yy_push_state(int new_state)
void yy_pop_state()
int yy_top_state()
```

Hiermit kann man eine **Sonderbehandlung innerhalb** einer **Sonderbehandlung** machen. Man hat die **Leistungsfähigkeit** von **Unterprogrammaufrufen** mit **Return**, im **Gegensatz** zu einfachen **Goto**s.

Dies ist allerdings **Flex-spezifisch**.

5.12 Flex und andere Lexer

Flex ist fast vollständig **POSIX-kompatibel**.

Einige **kleine technische Inkompatiblitäten mit** dem **originalen Lex** von AT&T.

AT&Ts Lex kennt keine **exklusiven Startbedingungen** mit `%x`, obwohl sie im POSIX-Standard stehen.

Flex schließt **Definitionen von regulären Ausdrücken** in **Klammern** ein, damit ein nachfolgender `*` oder `+` auf die ganze Definition wirkt. Das vermeidet Überraschungen. AT&Ts Lex tut das nicht. Mit der Option `-l` kann man bei Flex das **Lex-Verhalten einschalten**.

Es gibt ein paar ähnliche **Unterschiede bei regulären Ausdrücken**, wo Flex eine sinnvollere bzw. POSIX-kompatible Alternative wählt.

Flex hat eine ganze Reihe **eigener Erweiterungen**, die es nur dort gibt:

- Scanner optional in C++
- Alle `%option`
- Fast alle obigen Aufruf-Optionen.
- Buchstabenklassen
- **Lesen aus Strings** mit yy_scan_string() usw.
  (Andere Lexer haben jeweils **andere Wege** dafür.)
- **Verschachtelte Includes** mit yy_switch_to_buffer().
  (Andere Lexer haben jeweils **andere Wege** dafür.)
- `<<EOF>>` in Regeln.
- Bereiche von Startbedingungen
- Stacks von Startbedingungen
- ...