4.2 lex: Fortgeschrittenes

4.2 lex: Fortgeschrittenes

Aufbau einer lex-Datei

Definitionen
%%
Regeln
%%
Unterprogramme

} optional

Reguläre Ausdrücke von lex

- wichtigste Unterschiede zu sed/grep:
  - **`"while"`**: Literale können/sollten in **`""`** stehen
  - **`\n`** – Newline: (fast) normales Zeichen
    - **`[^abc]`** paßt auch auf Newline
      - ggf. Abhilfe: **`[^abc\n]`**
    - **`.`** (Punkt) paßt nicht auf Newline
  - kein Backslash vor **`+`**, **`?`**, **`{`**, **`}`**, **`(`**, **`)`**
  - kein **`\<`**, **`\>`**
  - kein **`\1`**, **`\2`**, ...

Aufbau einer Aktion

- Stück C-Code
  1. einzelne C-Anweisung (mit Semikolon)
  2. C-Block (in geschweiften Klammern)
     - dann auch über mehrere Zeilen

Weitere nützliche Konstrukte

- Abkürzungen für reguläre Ausdrücke
  - im Definitionsteil definieren
  - Benutzung: in geschweiften Klammern
- mehrere Ausdrücke für eine Aktion
  - „|“ anstelle einer Aktion
- Variable yytext
  - aktuelles Lexem
- Makro ECHO
  - druckt yytext

4.2 lex: Fortgeschrittenes

Scanner-Zustände, Grundlagen

- oft verschiedene Sätze von Mustern
  für verschiedene Abschnitte der Eingabe
  - Beispiele:
    - String: enthält niemals Schlüsselworte
    - Kommentar: ebenso
    - C-include-Pragmat: besondere String-Regeln für #include <stdio.h>

String-Regeln für C-include-Pragmat

- Strings entweder in
  - <, > oder
  - “, “

Demo

String-Regeln für C-include-Pragmat: Lösung

- cinclude-example.c
- cinclude.l

```
%{
#include <stdio.h>
%}
%option main
%x INCL
ID      [a-zA-Z_][a-zA-Z0-9_]*
%%
^"#include"[ \t]*   { printf("Pragmat #include erkannt.\n");
                      BEGIN(INCL);
                    }
<INCL>"<"[a-zA-Z._-]+">"   |
<INCL>"\""[a-zA-Z._-]+"\"" {
                      printf("Dateiname erkannt: '%s'\n", yytext);
                      BEGIN(INITIAL);
                    }
<INCL>.             printf("Illegales Zeichen nach '#include': '%s'\n", yytext);

{ID}                printf("Bezeichner erkannt: '%s'\n", yytext);
[ \t\n]+            /* ueberspringe White-Space */
.                   printf("Zeichen erkannt: '%s'\n", yytext);
```


Übung: C-Quellcode-Zähler

- liest C-Quelldatei und zählt, wieviel Zeilen mit:
  - C-Code
  - nur Kommentar
  - nur Whitespace
- Hinweise:
  - lies kleine Stücke
  - benutze Flags, ob schon C-Code oder Kommentar in dieser Zeile gesehen
  - bei Newline: inkrementiere Zähler, setze Flags zurück
  - Annahme: Strings enthalten keinen Kommentaranfang oder Newline

Demo

Mengen von Scanner-Zuständen

- Regel für *mehrere* Zustände:
  `<Z1,Z2,Z3>“muster“  { aktion(); }`
- Regel für *alle* Zustände:
  `<*>“muster“  { aktion(); }`

Übung: C-Quellcode-Zähler (2)

- Zusatzaufgaben:
  - Strings dürfen Kommentaranfang enthalten
  - Strings dürfen Newline enthalten

Demo

Umlenken der Ein- und Ausgabe

- Dateieingabe und -ausgabe umlenken
- Lesen aus Strings
- mehrere Eingabequellen nacheinander
- mehrere Eingabequellen abwechselnd

Dateieingabe und -ausgabe umlenken

- Datei-Zeiger `yyin`, `yyout`
- können normal zugewiesen werden

Demo

Dateieingabe und -ausgabe umlenken: Lösung

- username-yyin.l

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%option noyywrap
%%
"<username>"    printf("%s", getenv("USER"));
%%
int main(int argc, char *argv[]) {
    if (argc > 1) {
        yyin = fopen(argv[1], "r");
    }
    yylex();
}
```

Lesen aus Strings

- `yy_scan_string(const char *str)`
  - liest ab jetzt aus Null-terminiertem String `str`
- `yy_scan_bytes(const char *str, int len)`
  - liest `len` Bytes
- gilt nur für flex
  - andere Syntax bei anderen Versionen von lex

Kommandozeile auswerten

- Kommandozeile: Folge von Strings

Demo

Kommandozeile auswerten: Lösung

- 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);
}
```


Kommandozeile auswerten mit Scanner-Zuständen

- neue Option `-f` verlangt nachfolgenden Dateinamen

Demo

Kommandozeile auswerten mit Scanner-Zuständen: Lösung

- 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);
}
```


Mehrere Eingabequellen nacheinander

- bisher: neuer `yylex()`-Aufruf für jeden Kommandozeilenparameter ☹
- Funktion `yywrap()`
  - vom Benutzer geschrieben
  - wird am Ende der Eingabe automatisch aufgerufen
  - falls Rückgabewert 0: lex liest einfach weiter
    - in `yywrap()`: `yyin` bzw. String neu setzen
  - `%option noyywrap`
    - man muß kein `yywrap()` schreiben
    - (wird impliziert von `%option main`)

Demo

Mehrere Eingabequellen nacheinander: 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);
}
```


Mehrere Eingabequellen abwechselnd

- typische Anwendung: `include`
- Problem: Lexer liest voraus und puffert
  - geht hier nicht: `yyin` zuweisen
- Lösung: mehrere Eingabepuffer, Umschalten

Mehrere Eingabequellen abwechselnd (2)

- `YY_BUFFER_STATE yy_create_buffer(FILE *file, int size)`
  - erzeugt neuen Puffer
  - Empfehlung für `size` ist `YY_BUF_SIZE`
- `void yy_switch_to_buffer(YY_BUFFER_STATE new_buffer)`
  - schaltet zu neuem Puffer um
  - Scanner-„Zustand“ wird *nicht* verändert
- `void yy_delete_buffer(YY_BUFFER_STATE new_buffer)`
  - gibt Speicherplatz eines Puffers wieder frei

Mehrere Eingabequellen abwechselnd (3)

- `YY_CURRENT_BUFFER`
  - liefert aktuellen Puffer
  - Typ: `YY_BUFFER_STATE`
  - ist Makro

Demo

Mehrere Eingabequellen abwechselnd: Beispiel

- include-example[012].txt
- include.l

```
%{
#include <stdio.h>
#define MAX_INCLUDE_DEPTH 10
YY_BUFFER_STATE include_stack[MAX_INCLUDE_DEPTH];
int include_stack_pos = 0;
FILE *tmp_file_ptr = NULL;
%}
%option main
%option yywrap
%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(!(tmp_file_ptr = fopen(yytext, "r"))) {
                      fprintf(stderr,
                        "Cannot open include file '%s'!\n", yytext);
                      exit(1);
                    }
                    yy_switch_to_buffer(
                      yy_create_buffer(tmp_file_ptr, YY_BUF_SIZE));
                    BEGIN(INITIAL);
                  }
%%
int yywrap() {
    if(--include_stack_pos < 0)
        return 1;
    else {
        yy_delete_buffer(YY_CURRENT_BUFFER);
        yy_switch_to_buffer(
          include_stack[include_stack_pos]);
        return 0;
    }
}
```


4.2 lex: Fortgeschrittenes

Vordefinierte Buchstabenmengen in Mustern

- `[:alnum:] [:alpha:] [:digit:]`
  `[:blank:] [:space:]`
  `[:lower:] [:upper:]`
  ...
- Beispiel: `[[:alpha:]0-9]+`
- definiert über Funktionen der C-Bibliothek `isalnum()`, ...
  - siehe `man isalnum`
- ist flex-Erweiterung

Muster für Dateiende

- `<<EOF>>`
  - paßt genau auf Dateiende
  - kann nicht mit anderen Zeichen kombiniert werden
  - ist Erweiterung zu `yywrap()`
    - `<<EOF>>`-Regel darf von Scanner-Zustand abhängen
  - ist flex-Erweiterung

Vorschau-Operator

- Beispiel: `“include “/\“[^“\n]*\“`
  - das gematchte Muster ist `“include “`
  - aber es muß ein Dateiname in Anführungstrichen folgen
- bereits bekannter Spezialfall: `$`
  - Vorschau auf Zeilenwechsel

4.2 lex: Fortgeschrittenes

Bereiche für Scanner-Zustände

- Problem:
  oft viele Regeln mit gleichem Scanner-Zustand
  <Z1>Muster Aktion
  <Z1>...
  <Z1>Muster Aktion
- Lösung:
  <Z1>{
  Muster Aktion
  ...
  Muster Aktion
  }
  - ist flex-Erweiterung

Stack von Scanner-Zuständen

- Problem:
  manchmal reicht endlicher Automat
  von Scanner-Zuständen nicht aus
  - Rücksprung zu gemerktem Zustand nötig
  - Beispiel:
    Sonderbehandlung innerhalb einer Sonderbehandlung
- Lösung:
  - `%option stack`
  - `void yy_push_state(int new_state)`
  - `void yy_pop_state()`
  - `int yy_top_state()`
- ist flex-Erweiterung

Exklusive/inklusive Scanner-Zustände

- `%x Z1 Z2 Z3`
  - „exklusiver Scanner-Zustand“
  - nach `BEGIN(Z2)`: nur noch Regeln für `Z2` aktiv
  - ist flex-Erweiterung
- `%s Z1 Z2 Z3`
  - „inklusiver Scanner-Zustand“
  - nach `BEGIN(Z2)`: Regeln für `Z2` *und* Regeln ohne angegebenen Scanner-Zustand aktiv
  - ist ein Design-Fehler des Standard-lex

4.2 lex: Fortgeschrittenes

Mehrere Lexer in einem Programm

- Problem:
  alle globalen Funktionen und Variablen von lex haben festen Namen: `yylex()`, `yytext`, ...
- Lösung:
  - Kommandozeile: `-Pfoo`
  - `%option prefix="foo"`
  - ergibt `foolex()`, `footext`, ...
  - anderes Präfix für jeden der Lexer nehmen

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`	Gib die möglichen Aufrufoptionen aus.
`%option yylineno`		Die Variable `yylineno` enthält immer die aktuelle Zeilennummer.

Weitere Aufruf- und Datei-Optionen

Datei	Aufruf	Bedeutung
`%option prefix="XYZ"`	`-PXYZ`	Ersetze in allen Variablen- und Funktionsnamen das Präfix yy durch XYZ.
`%option stack`		Erlaube Stacks von Startbedingungen.
`%option c++`	`-+`	Generiere eine C++-Scanner-Klasse.
`%option perf-report`	`-p`	Gib einen Performance-Report auf `stdout` aus, der „teure" benutzte Features auflistet.
`%option lex-compat`	`-l`	Maximale Kompatibilität mit originalem lex von AT&T.
`%option debug`	`-d`	Generierter Scanner macht Debug-Ausgaben, wenn Variable `yy_flex_debug` ≠ 0.
	`--version`	Gib die Versionsnummer aus.
...	...	...

flex und andere Lexer

- flex
  - fast vollständig POSIX-kompatibel
  - einige kleine technische Inkompatibilitäten mit originalem lex von AT&T
    - siehe Manual von lex
- lex von AT&T
  - keine exklusiven Scanner-Zustände **%x**, obwohl in POSIX

Erweiterungen in flex

- alle `%option`
- fast alle obigen Aufruf-Optionen
- Buchstabenklassen, z.B. `[:alnum:]`
- Bereiche von Scanner-Zuständen
- Stacks von Scanner-Zuständen
- `<<EOF>>` in Mustern
- die Syntax für Lesen aus Strings
- die Syntax für Lesen aus verschiedenen Puffern
- Scanner optional in C++
- ...

Inhalte der Vorlesung

1. Einführung
2. Lexikalische Analyse
3. Der Textstrom-Editor sed
4. Der Scanner-Generator lex
5. → Syntaxanalyse und der Parser-Generator yacc
6. Syntaxgesteuerte Übersetzung
7. Übersetzungssteuerung mit make