CSE216 Programming Abstractions

Scanner

YoungMin Kwon

Lexical Analysis

- Scanner
  - Read the input characters (usually from files)
  - Produce a sequence of tokens for a parser
    - Input char $\rightarrow$ token: reduce the # of items to handled by a parser
  - Remove comments
  - Save texts of interesting tokens
  - Remove white spaces

Files in Unix

- A Linux file is a sequence of bytes
  - $B_0, B_1, \ldots, B_k, \ldots, B_m$
- All I/O devices are modeled as *files*
  - E.g. networks, disks, terminals
  - Input and output are performed by reading from and writing to the appropriate files
  - This mapping enables simple low level APIs known as Unix I/O

File I/O

- Opening files
  - Announce an app's intention to access an I/O device.
  - Kernel returns a descriptor
- Changing the current file position
  - A byte offset from the beginning of a file
  - The kernel maintains a file position for each open file
  - *seek* operation can change the file position

File I/O

- Reading and writing files
  - *read* copies n bytes from the current position of a file to memory
  - *write* copies n bytes from memory to the current position of a file
- Closing files
  - Informs the kernel that the app has finished accessing the file

Some Functions for File I/O

```c
#include <stdio.h>

//open a file for reading and/or writing
FILE *fopen(const char *pathname, const char *mode);

//close a file
int fclose(FILE *stream);

//write a formatted string to a file
int fprintf(FILE *stream, const char*format, ...);

//read a formatted data from a file
int fscanf(FILE *stream, const char *format, ...);
```

Some Functions for File I/O

```
//read nmemb elements of size data from file
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);

//write nmemb elements of size data to file
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);

//change the current position of the file
//   whence can be SEEK_SET, SEEK_CUR, or SEEK_END
extern int fseek(FILE *stream, long offset, int whence);
```

File I/O

```
FILE *fopen(const char *pathname, const char *mode);
/*
pathname: path to the file to open
mode:
r   Open text file for reading.
    The stream is positioned at the beginning of the file.

r+  Open for reading and writing.
    The stream is positioned at the beginning of the file.

w   Truncate file to zero length or create text file for writing.
    The stream is positioned at the beginning of the file.

w+  Open for reading and writing. The file is created if it does not exist, otherwise
    it is truncated. The stream is positioned at the beginning of the file.

a   Open for appending (writing at end of file). The file is created
    if it does not exist.  The stream is positioned at the end of the file.

a+  Open for reading and appending (writing at end of file).
    The file is created if it does not exist. Output is always appended to
    the  end of the file.

b   add b at the end or after the first character for binary files
*/
```

Hello.c

```c
#include <stdio.h>

int main() {
    FILE *fp;
    //open hello.txt for write mode
    fp = fopen("hello.txt", "w");

    //write a string to the file
    fprintf(fp, "Hello world!\n");

    //close the file
    fclose(fp);

    return 0;
}
```

```
> dir /w hello.*
…
hello.c     hello.txt
…

> type hello.txt
Hello world!
```

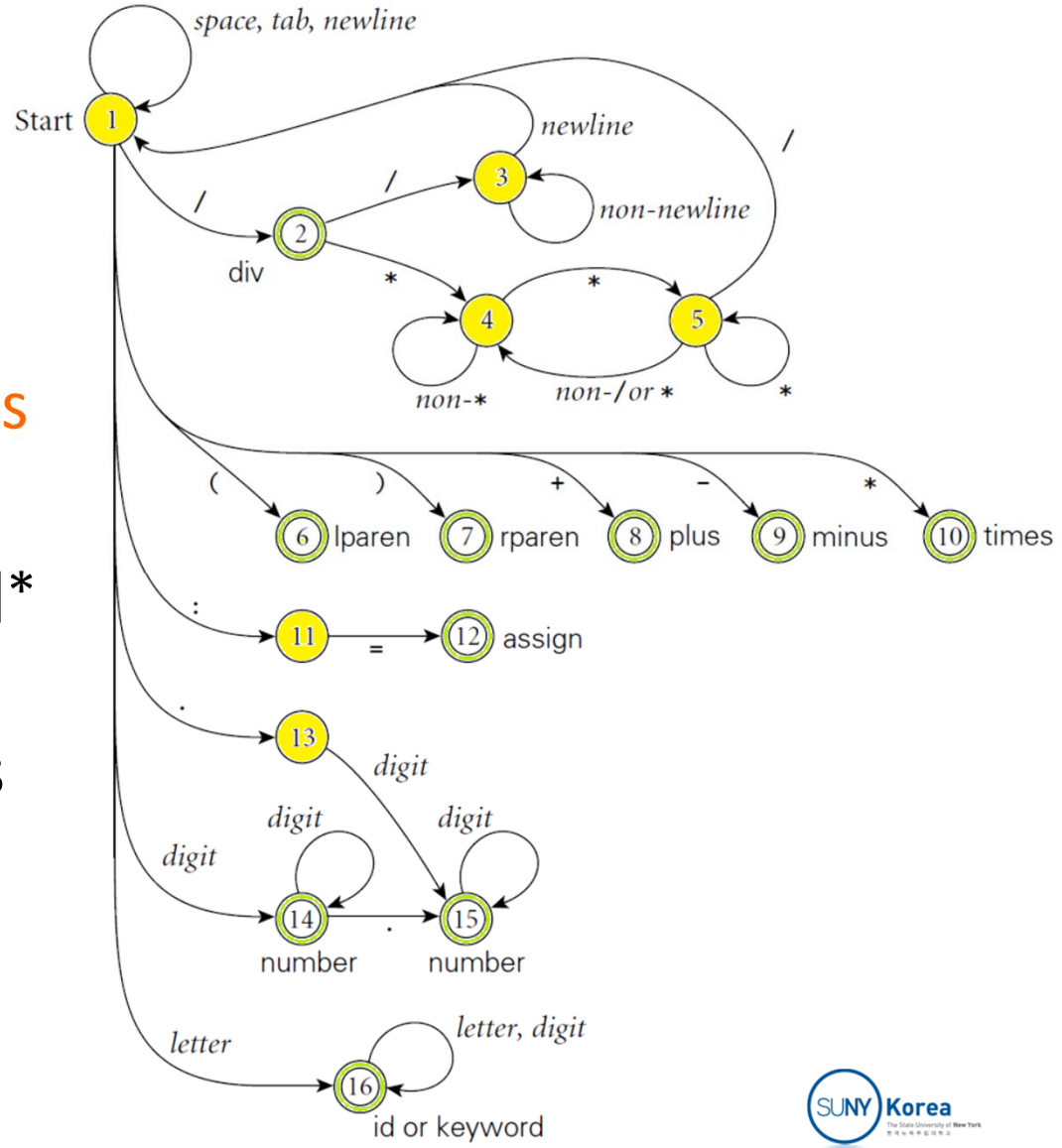
Example: copy files

```
//
// dupe.c
//
#include "common.h"
#include <stdio.h>

void dupe(FILE *src, FILE *dst) {
    char buf[256];
    size_t n;
    while((n = fread(buf, 1, sizeof(buf), src)) > 0)
        fwrite(buf, 1, n, dst);
}
```

```c
int main(int argc, char** argv) {
    if(argc != 3) {
        printf("usage: dupe src_file dst_file\n");
        goto out;
    }
    FILE *src = fopen(argv[1], "r");
    ON_FALSE_GOTO(src != NULL, out, strmsg("cannot open %s", argv[1]));

    FILE *dst = fopen(argv[2], "w");
    ON_FALSE_GOTO(dst != NULL, out, strmsg("cannot open %s", argv[2]));

    dupe(src, dst);

    fclose(src);
    fclose(dst);

out:
    return 0;
}
```

Finite Automata

- Scanner
  - Tokens are described in regular expressions
    - [0-9]+
    - [a-zA-Z_][a-zA-Z_0-9]*
  - A finite automaton is built to recognize regular expressions



Regular Expressions

- Scanner
  - Tokens are described in regular expressions
  - A finite automaton is built for the regular expressions

assign ⟶ `:=`
plus ⟶ `+`
minus ⟶ `-`
times ⟶ `*`
div ⟶ `/`
lparen ⟶ `(`
rparen ⟶ `)`
id ⟶ *letter* (*letter* | *digit*)*
except for `read` and `write`
number ⟶ *digit digit** | *digit** (. *digit* | *digit* .) *digit**

comment ⟶ `/*` (*non-** | `*` *non-/*)* `*/`
| `//` (*non-newline*)* *newline*

Regular Expressions

- Operators
  - `.` matches any single character
  - `[]` matches any characters in the bracket
  - `[^]` matches any characters except for the ones in the bracket
  - `^` matches beginning of a line
  - `$` matches end of a line
  - `\` escape sequence

Regular Expressions

- Operators (cont'd)
  - | matches either the preceding expr or the following expr
  - * matches zero or more copies of the preceding expression
  - + matches one or more copies of the preceding expression
  - ? matches zero or one occurrence of the preceding expression
  - ( ) groups a series of expressions

Regular Expressions

- Examples
  - Single digit: `[0-9]`
  - Integer: `[0-9]+`
  - Integer with optional minus sign: `-?[0-9]+`
  - Float: `[0-9]*(.[0-9]|[0-9].)[0-9]*`
  - Identifier: `[a-zA-Z_][a-zA-Z_0-9]*`

Regex

- POSIX standard for regular expressions

```
//
//in regex.h
//

extern int regcomp (regex_t *preg, const char *pattern, int cflags);

extern int regexec (const regex_t *preg, const char *string,
                    size_t nmatch, regmatch_t pmatch[], int eflags);

extern size_t regerror (int errcode, const regex_t *preg,
         char *errbuf, size_t errbuf_size);

extern void regfree (regex_t *preg);
```

```c
#include "regex.h"
#include "common.h"
#include <stdio.h>

enum tokens {
    TOK_ALL = 0,    //match for the whole expr
    TOK_WS = 1,     //white space
    TOK_NUM,        //number
    TOK_KEY,        //keywords
    TOK_ID,         //identifier
    TOK_PUN,        //punctuator
    TOK_COUNT,      //token count
};

#define TOK_REGEXP  "^([ \t\n\r]+)|"                /*whitespace*/\
                    "^([0-9]+)|"                    /*number*/\
                    "^(fun|if|then|else)|"          /*keywords*/\
                    "^([_a-zA-Z][_a-zA-Z0-9]*)|"    /*identifier*/\
                    "^(->|=|<>|<=|<|>=|>)"          /*punctuator*/
```

```
void init_regexps(regex_t *regTokens) {
    int res = regcomp(regTokens, TOK_REGEXP, REG_EXTENDED);
    if(res != 0) {
        char msg[256];
        regerror(res, regTokens, msg, sizeof(msg));
        ON_FALSE_EXIT(0, strmsg("error in regcomp: %d, %s", res, msg));
    }
}

static char* token_text(char *str, regmatch_t *match) {
    static char text[256];
    int i = 0, j = match->rm_so;
    while(j < match->rm_eo && i < sizeof(text) - 1)
        text[i++] = str[j++];
    text[i] = 0;
    return text;
}
```

```
void scan(regex_t *regTokens, char *str) {
    char text[256];
    regmatch_t matches[TOK_COUNT];
    char *s = str;
    while(*s) {
        int res = regexec(regTokens, s, TOK_COUNT, matches, 0);
        if(res == 0) {
            if(matches[TOK_WS].rm_so == 0)                  //white space
                s += matches[TOK_WS].rm_eo;
            else if(matches[TOK_NUM].rm_so == 0) {          //number
                printf("%s:num ", token_text(s, &matches[TOK_NUM]));
                s += matches[TOK_NUM].rm_eo;
            }
            else if(matches[TOK_KEY].rm_so == 0) {          //keywords
                printf("%s:key ", token_text(s, &matches[TOK_KEY]));
                s += matches[TOK_KEY].rm_eo;
            }
            else if(matches[TOK_ID].rm_so == 0) {           //identifier
                printf("%s:id ", token_text(s, &matches[TOK_ID]));
                s += matches[TOK_ID].rm_eo;
            }
            else if(matches[TOK_PUN].rm_so == 0) {          //punctuator
                printf("%s:pun ", token_text(s, &matches[TOK_PUN]));
                s += matches[TOK_PUN].rm_eo;
            }
        }
…
```

```
…
        else if(res == REG_NOMATCH) {
            printf("%c ", *s);
            s++;
        }
        else {
            char msg[256];
            regerror(res, regTokens, msg, sizeof(msg));
            ON_FALSE_EXIT(0, strmsg("error in regexec: %d, %s", res, msg));
        }
    }
}

int main() {
    char *str = "fun x ->          \n"
                "    fun y ->      \n"
                "        if x > y  \n"
                "        then x    \n"
                "        else y"
            ;
    regex_t regTokens;
    init_regexps(&regTokens);

    printf("Lexical analysis:\n%s\n", str);
    scan(&regTokens, str);

    regfree(&regTokens);
}
```

Result

```
> a.exe
Lexical analysis:
fun x ->
    fun y ->
        if x > y
        then x
        else y
fun:key x:id ->:pun fun:key y:id ->:pun if:key x:id >:pun y:id
then:key x:id else:key y:id
```

To Compile

- In Windows:

```
> where gcc
C:\Program Files (x86)\mingw-w64\
i686-8.1.0-posix-dwarf-rt_v6-rev0\mingw32\bin\gcc.exe

Copy …\mingw32\opt\include\regex.h to your local directory
Copy …\mingw32\opt\lib\libregex.a  to your local directory

> gcc scanner.c common.c libregex.a
```

- In Linux and Mac:

```
$ gcc scanner.c common.c
```

Exercise

- Let's handle the C style single line comments
  - Ignore // to the end of the line (\n)

```
int main() {
    char *str = "fun x ->          //take the first param\n"
                "    fun y ->      //take the second param\n"
                "        if x > y //return the larger of the two\n"
                "        then x\n"
                "        else y"
            ;
    regex_t regTokens;
    init_regexps(&regTokens);

    printf("Lexical analysis:\n%s\n", str);
    scan(&regTokens, str);

    regfree(&regTokens);
}
```

A Solution

```
enum tokens {
    TOK_ALL = 0,     //match for the whole expr
    TOK_WS = 1,      //white space
    TOK_NUM,         //number
    TOK_KEY,         //keywords
    TOK_ID,          //identifier
    TOK_PUN,         //punctuator
    TOK_COMMENT,     //comment
    TOK_COUNT,       //token count
};

#define TOK_REGEXP   "^([ \t\n\r]+)|"                 /*whitespace*/\
                     "^([0-9]+)|"                     /*number*/    \
                     "^(fun|if|then|else)|"           /*keywords*/  \
                     "^([_a-zA-Z][_a-zA-Z0-9]*)|"     /*identifier*/\
                     "^(->|=|<>|<=|<|>=|>)|"          /*punctuator*/\
                     "^(//[^\n]*\n)"                  /*comment*/
```

A Solution

```
void scan(regex_t *regTokens, char *str) {
    char text[256];
    regmatch_t matches[TOK_COUNT];
    char *s = str;
    while(*s) {
        int res = regexec(regTokens, s, TOK_COUNT, matches, 0);
        if(res == 0) {
…
            else if(matches[TOK_COMMENT].rm_so == 0) { //comment
                s += matches[TOK_COMMENT].rm_eo;
            }
        }
…
```

Course Evaluation

Your feedback is important!

- Please submit your Course Evaluation at https://stonybrook.campuslabs.com/eval-home/
- Course Outcome Survey for ABET at https://forms.gle/YAZag8ccNi1wgFyC6

SUNY Korea