Lexical Analysis

Regular Expressions

Nondeterministic Finite Automata (NFA)

Deterministic Finite Automata (DFA)

Implementation Of DFA

Key Differences for a Scanner and RE Recognizer

- Given a single string, automata and regular expressions returned a Boolean answer:
  - a given string is/is not in a language

- In contrast ...
- Given an input (an EOF-terminated “long” string), a scanner returns a series of tokens
  - finds the longest lexeme, and
  - returns the corresponding token
A Sample Scanner

- The language of assignment statements:
  - LHS = RHS
  - LHS = RHS
  - ...

  - left-hand side of assignment is a simple identifier:
    - a letter followed by one or more letters or digits
  - right-hand side is one of the following:
    - ID + ID
    - ID * ID
    - ID == ID

---

Step 1: Define tokens

- Our language has five tokens,
  - They can be defined by five regular expressions:

<table>
<thead>
<tr>
<th>Token</th>
<th>Regular Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Step 2: Convert REs to NFAs

ASSIGN:
ID:
PLUS:
TIMES:
EQUALS:

Step 3: Convert NFAs to DFAs

ASSIGN:
ID:
PLUS:
TIMES:
EQUALS:
Step 4: Combining per-token DFAs

Goal of a scanner:
- find the *longest prefix* of the current input that corresponds to a token.

This has two consequences:

- lookahead:
  - Examine if the next input character can "extend" the current token. If yes, keep building a larger token.

- a real scanner cannot get stuck:
  - What if we get stuck building the larger token?
    Solution: return characters back to input.

Furthermore ...

- In general the input can correspond to a series of tokens (lexemes), not just a single token
  - **Problem**: It is no longer correct to run the FSM until it gets stuck or whole string is consumed.
    So, how to partition the input into lexemes?
  - **Solution**: a token must be returned when a regular expression is matched

- Some lexemes (like whitespace and comments) do not correspond to tokens
  - **Problem**: how to “discard” these lexemes?
  - **Solution**: after finding such a lexeme, the scanner simply starts again and tries to match another regular expression
Extend the DFA

- modify the DFA so that an edge can have
  - an associated action to
    - "put back one character" or
    - "return token XXX",
  - such DFA is called a transducer

- we must combine the DFAs for all of the tokens into a single DFA

Step 4: Example of extending the DFA

- The DFA that recognizes simple identifiers must be modified as follows:

  - letter | digit
  - action:
    - put back 1 char
    - return ID
  - any char except letter or digit

- recall that scanner is called by parser
  (one token is return per each call)
- hence action return puts the scanner into state S
Implementing the extended DFA

- The table-driven technique works, with a few small modifications:
  - Include a column for end-of-file
    - e.g., to find an identifier when it is the last input token
  - besides 'next state', a table entry includes
    - an (optional) action: put back $n$ characters, return token
  - Instead of repeating
    - "read a character; update the state variable" until the machine gets stuck or the entire input is read,
    - "read a character; update the state variable; perform the action"
  - eventually, the action will be to return a value, so the scanner code will stop

---

Step 4: Example: Combined DFA for our language

![Diagram of DFA network for language example]
Scanner Transition Table

<table>
<thead>
<tr>
<th>Input/State</th>
<th>+</th>
<th>*</th>
<th>=</th>
<th>Letter</th>
<th>Digit</th>
<th>EOF</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>NoA</td>
<td>NoA</td>
<td>NoA</td>
<td>NoA</td>
<td>NoA</td>
<td></td>
</tr>
<tr>
<td>Final</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Plus</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Final</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Times</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TMP</td>
<td>PB1</td>
<td>PB1</td>
<td>PB1</td>
<td>NoA</td>
<td>NoA</td>
<td>PB1</td>
</tr>
<tr>
<td>Final</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ident</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PB1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Final</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ident</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PB1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PB1</td>
</tr>
<tr>
<td>Final</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assn</td>
<td>PB1</td>
<td>PB1</td>
<td>PB1</td>
<td>PB1</td>
<td>PB1</td>
<td>PB1</td>
</tr>
<tr>
<td>Final</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PB1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Final</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PB1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Final</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assn</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Actions: (Green)
- NoA – no action
- PB1 – put back 1 char

States (Blue)

Result Tokens (Red)

Possible Extensions

Augment "combined" finite-state machine to:

- Ignore white-spaces between tokens
  - white-spaces are spaces, tabs and newlines
- Give an error message if
  - a character other than +, *, =, letter, or digit occurs in the input, or
  - a digit is seen as the first character in the current input
  - (in both cases, ignore the bad character)
- Return an EOF token when there are no more tokens in the input
### Updated Transition Table

<table>
<thead>
<tr>
<th>Input/State</th>
<th>+</th>
<th>*</th>
<th>=</th>
<th>Letter</th>
<th>Digit</th>
<th>EOF</th>
<th>WS</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>S</strong></td>
<td>NoA</td>
<td>NoA</td>
<td>NoA</td>
<td>NoA</td>
<td>Error</td>
<td>NoA</td>
<td>NoA</td>
<td>Error</td>
</tr>
<tr>
<td></td>
<td>Final</td>
<td>Final</td>
<td>TMP</td>
<td>ID</td>
<td>-</td>
<td>-</td>
<td>S</td>
<td>-</td>
</tr>
<tr>
<td><strong>ID</strong></td>
<td>PB1</td>
<td>PB1</td>
<td>PB1</td>
<td>NoA</td>
<td>NoA</td>
<td>PB1</td>
<td>PB1</td>
<td>PB1</td>
</tr>
<tr>
<td></td>
<td>Final</td>
<td>Final</td>
<td>Final</td>
<td>ID</td>
<td>-</td>
<td>Final</td>
<td>Final</td>
<td>Final</td>
</tr>
<tr>
<td><strong>TMP</strong></td>
<td>PB1</td>
<td>PB1</td>
<td>PB1</td>
<td>NoA</td>
<td>PB1</td>
<td>PB1</td>
<td>PB1</td>
<td>PB1</td>
</tr>
<tr>
<td></td>
<td>Final</td>
<td>Final</td>
<td>Final</td>
<td>Assn</td>
<td>Final</td>
<td>Final</td>
<td>Final</td>
<td>Assn</td>
</tr>
</tbody>
</table>

### Implementation in Flex

```flex
#include "assign_tokens.h"

DIGIT [0-9]
LETTER [A-Za-z]

#define T_EOF 1
#define T_PLUS 11
#define T_TIMES 12
#define T_EQUALS 13
#define T_ASSIGN 21
#define T_IDENT 31

extern int char_num;
extern int line_num;
extern int s_char_num;
extern int s_line_num;

%{
#include "assign_tokens.h"
%

DIGIT [0-9]
LETTER [A-Za-z]

%{
#define T_EOF 1
#define T_PLUS 11
#define T_TIMES 12
#define T_EQUALS 13
#define T_ASSIGN 21
#define T_IDENT 31

extern int char_num;
extern int line_num;
extern int s_char_num;
extern int s_line_num;

`include "assign_tokens.h"
``
```
#include <iostream.h>
#include <stdio.h>
#include "assign_tokens.h"

extern FILE *yyin;
extern char *yytext;
extern int yylex();

int char_num = 1;
int line_num = 1;
int s_char_num;
int s_line_num;

int main (int argc, char *argv[]) {
if (argc != 2) {
    cout <<
        "Usage: assign_scan <file>" << endl;
    return -1;
}
if ((yyin = fopen(argv[1],"r")) == NULL) {
    cout <<
        "Error! Unable to open file " << argv[1] << endl;
    return -1;
}

int token_num;
do {
    s_char_num = char_num;
    s_line_num = line_num;
    token_num = yylex();
    cout << "Token: ";
    switch (token_num) {
        case T_EOF: cout << "; break;
        case T_PLUS: cout << "+"; break;
        case T_TIMES: cout << "*"; break;
        case T_EQUALS: cout << "; break;
        case T_ASSIGN: cout << "="; break;
        case T_IDENT: cout << "Ident(" << yytext << "); break;
    }
    cout << "; found starting at line " << s_line_num << "; break;
    cout << "; char " << s_char_num << "; break;
} while (token_num != T_EOF);
return 0;
}
Implementing Table Machine

- Need input stream with the ability to put back characters (standard in many C++ streams)
- Need table entries for every state/character combination
- Table entries should indicate (1) an action, (2) a resulting state, and (3) a return token (if any)

Adding PutBack to a Stream

```c
#define MY_EOF 256
bool putback_char_p = 0;
int putback_char;
int prev_char_num;
int prev_line_num;
int char_num = 1;
int line_num = 1;
int next_char(ifstream &inf){
    int ch;
    if (putback_char_p) {
        putback_char_p = 0;
        ch = putback_char; }
    else ch = inf.get();
    if (ch == EOF) ch = MY_EOF;
    prev_char_num = char_num;
    prev_line_num = line_num; }
```
Implementing a State Machine

```c
#define No_State        0
#define Start_State     1
#define ID_State        2
#define TMP_State       3
#define Final_State     4
#define T_EOF           1
#define T_PLUS          11
#define T_TIMES         12
#define T_EQUALS        13
#define T_ASSIGN        21
#define T_IDENT         31
#define No_Action       0
#define PutBack_1       1
#define Report_Error    2

int txtloc = 0;
char yytext[BUFFERSIZE];

class TableEntry {
    public:
        int raction;
        int rstate;
        int rtoken;

    TableEntry {
        int raction;
        int rstate;
        int rtoken;
    }
}

stab[FINAL_STATE+1][MY_EOF+1];
```

```c
ifstream yyin;

int yylex () {
    int restok, ch;
    int currs = Start_State;
    txtloc = 0;
    yytext[txtloc] = '\0';
    s_line_num = line_num;
    s_char_num = char_num;
    do {
        ch = next_char(yyin);
        int a =
            stat[currs][ch].raction;
        switch (a) {
            case No_Action:
                txtloc++;
                yytext[txtloc]=(char)ch;
                break;
            case PutBack_1_Action:
                put_back_char(ch);
                break;
            case Report_Error_Action:
                cout <<
                    "Error! Unexpected char (" << (char) ch << ") at line " << prev_line_num << ", char " << prev_char_num << "!
                " << endl;
                break;
        }
    }
    return restok;
}
```

Implementing a State Machine

```c
restok = stab[currs][ch].rtoken;
currs = stab[currs][ch].rstate;
if (currs == Start_State) {
    yytext_loc = 0;
    yytext[yytext_loc] = '\0';
    s_line_num = line_num;
    s_char_num = char_num;
}
} while (curr_state != Final_State);

return result_token;
```

Changes to main?

- Call to initialize scanner table

**Hand Crafting a Scanner – Identify and Finish**

```c
int yylex () {
    int ch;
    // Define yytext
    do {
        // Update start line numbers
        ch = next_char(yyin);
        if (letter_p(ch)) {
            // add ch to yytext
            do {
                ch = next_char(yyin);
                if (/* ch let,dig */) /* add ch to yytext */
            } else {
                put_back_char(ch);
                return T_IDENT;
            }
        } else if (ch == MY_EOF)
            return T_EOF;
        else {
            switch ((char) ch) {
                case '+': return T_PLUS;
                case '*': return T_TIMES;
                case ' ': case '	': case '
': break;
                case '=': ch = next_char(yyin);
                    if (ch == '=')
                        return T_EQUALS;
            }
            return T_ASSIGN;
        }
    } while (1);
}
```
default: cout << 
"Error! Unexpected char (" << (char) ch << ") at line " << prev_line_num << ", char " << prev_char_num << ":!" << endl;
} 
} 
} while (1);
}