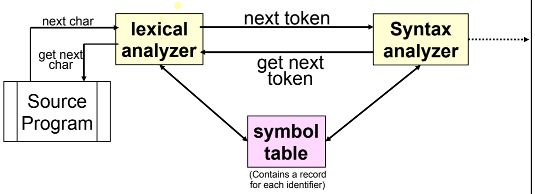
Compilers

Prof. Mohamed Hamada

Software Engineering Lab.
The University of Aizu
Japan

Lexical Analyzer (Scanner)

1. **Uses Regular Expressions to define tokens**
2. **Uses Finite Automata to recognize tokens**



token: smallest meaningful sequence of characters of interest in source program

What is tokens?

- Source program text → Tokens

- Examples Tokens
  - Operators = + - > ({ := == <>
  - Keywords if while for int double
  - Identifiers such as pi in program fragment const pi=3.14;
  - Numeric literals 43 6.035 -3.6e10 0x13F3A
  - Character literals ‘a’ ‘~’ ‘\’ ’
  - String literals “6.891” “Fall 98” “\”\” = empty”
  - Punctuation symbols such as comma and semicolon etc.
- Example of non-tokens
  - White space space(‘ ’) tab(‘\t’) end-of-line(‘\n’)
  - Comments /*this is not a token*/

How the scanner recognizes a token?

General approach:

1. Build a deterministic finite automaton (DFA) from regular expression *E*

2. Execute the DFA to determine whether an input string belongs to *L*(*E*)

Note: The DFA construction is done automatically by a tool such as ***lex***.

Regular Definitions

Regular definitions are regular expressions associated with suitable names.

For Example the set of identifiers in Java can be expressed by the Following regular definition:

letter → A | B | … | Z | a | b | … | z
digit → 0 | 1 | 2 | … | 9
id → letter (letter | digit)*

Regular Definitions

Notations

1. The `+` symbol denotes *one or more instance*
2. The `?` symbol denotes *zero or one instance*
3. The `[ ]` symbol denotes *character classes*

Example: **the following regular definitions represents unsigned numbers in C**

digit → [0 – 9]
digits → digit^{+}
fraction → (.digits)?
exponent → (E(+|-)? digits)?
number → digits fraction exponent

How to Parse a Regular Expression?

Given a DFA, we can generate an automaton that recognizes the longest substring of an input that is a valid token.

Using the three simple rules presented, it is easy to generate an NFA to recognize a regular expression.

Given a regular expression, how do we generate an automaton to recognize tokens?

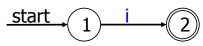
Create an NFA and convert it to a DFA.

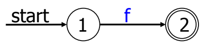
Regular expressions for some tokens

if	{return IF;}
[a - z] [a - z0 - 9] *	{return ID;}
[0 - 9] +	{return NUM;}
([0 - 9] + "." [0 - 9] *) | ([0 - 9] * "." [0 - 9] +)	{return REAL;}
("--" [a - z]* "\n") | (" " | "\n " | "\t ") +	{/* do nothing*/}
.	{error ();}

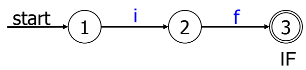
Building Finite Automata for Lexical Tokens

if	{return IF;}

The NFA for a symbol i is: 

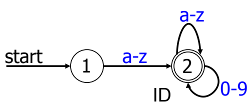
The NFA for a symbol f is: 

The NFA for the regular expression if is:



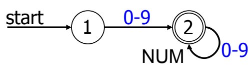
Building Finite Automata for Lexical Tokens

[a-z] [a-z0-9] *	{return ID;}



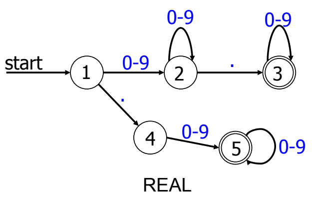
Building Finite Automata for Lexical Tokens

[0 - 9] +	{return NUM;}



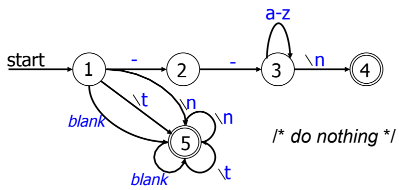
Building Finite Automata for Lexical Tokens

([0 - 9] +"."[0 - 9] *) | ([0 - 9] * "."[0 - 9] +) {return REAL;}

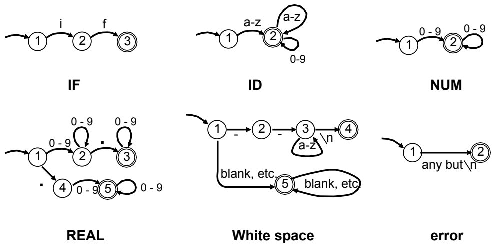


Building Finite Automata for Lexical Tokens

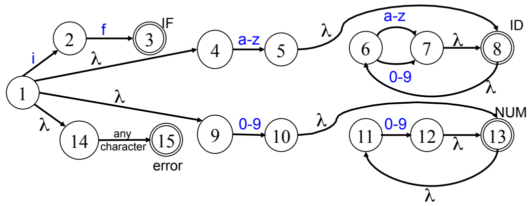
("--" [a - z]* "\n") | (" " | "\n " | "\t ") + {/* do nothing*/}



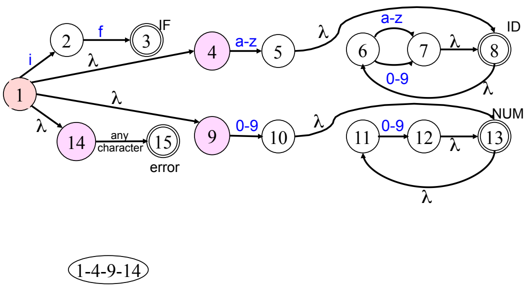
Building Finite Automata for Lexical Tokens



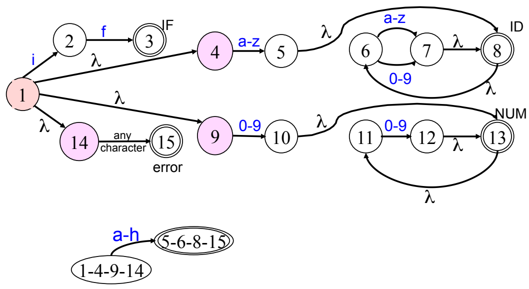
Conversion of NFA into DFA



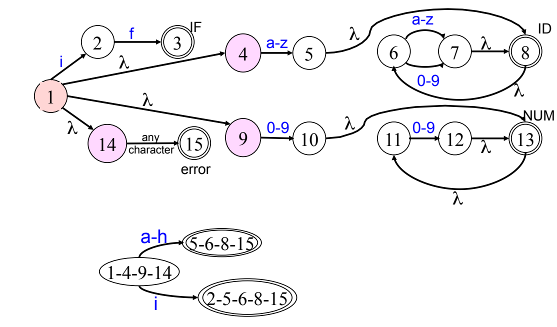
Conversion of NFA into DFA



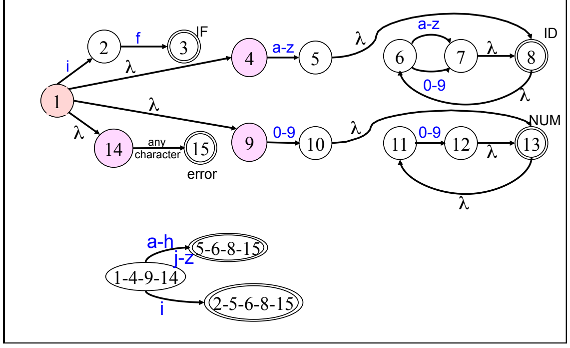
Conversion of NFA into DFA



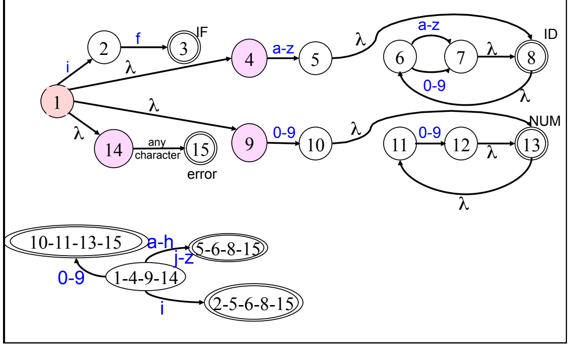
Conversion of NFA into DFA



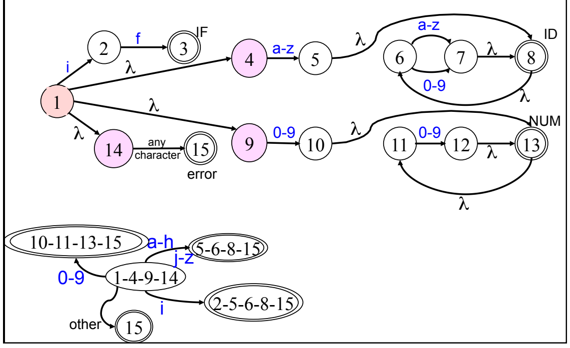
Conversion of NFA into DFA



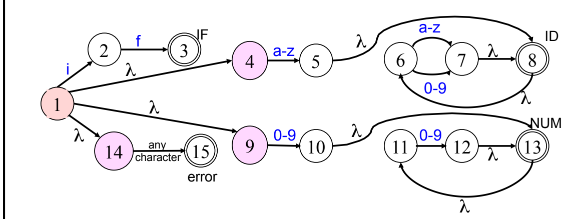
Conversion of NFA into DFA

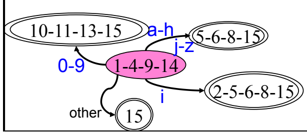


Conversion of NFA into DFA



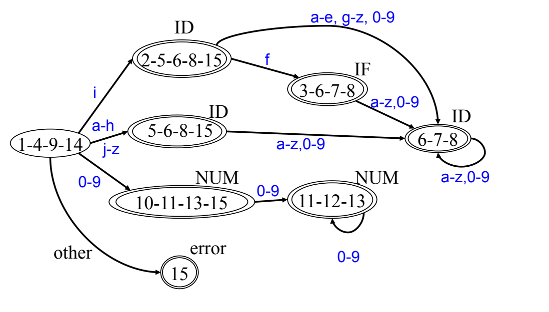
Conversion of NFA into DFA

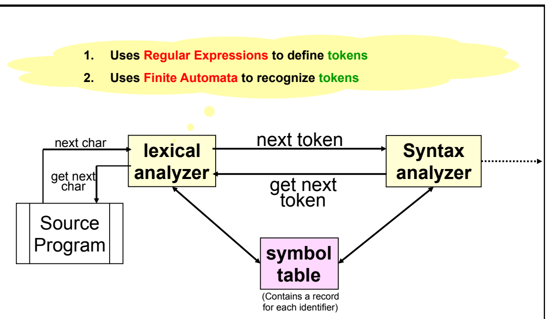




The analysis for 1-4-9-14 is complete. We mark it and pick another state in the DFA to analyze.

The corresponding DFA





token: smallest meaningful sequence of characters of interest in source program

How to write a scanner?

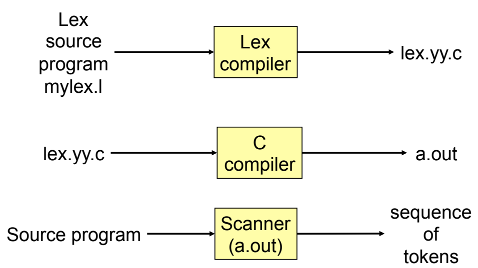
General approach:
The construction is done automatically by a tool such as the *Unix* program ***lex***.

Using the source program language grammar to write a simple lex program and save it in a file named lex.l

Using the unix program lex to compile to compile lex.l resulting in a C (scanner) program named lex.yy.c

Compiling and linking the C program lex.yy.c in a normal way resulting the required scanner.

Using Lex



Lex

In addition to compilers and interpreters, Lex can be used in many other software applications:

1. The desktop calculator *bc*
2. The tools *eqn* and *pic* (used for mathematical equations and complex pictures)
3. *PCC* (*P*ortable *C* *C*ompiler) used ith many UNIX systems
4. *GCC* (*G*NU *C* *C*ompiler) used ith many UNIX systems
5. A menu compiler
6. A SQL data base language syntax checker
7. The Lex program itself

And many more

Lex program specification

A Lex program consists of the following three parts:

declarations
%%
translation rules
%%
user subroutines (auxiliary procedures)

The first %% is required to mark the beginning of the translation rules and the second %% is required only if user subroutines follow.

Lex program specification

Declarations: include variables, constants and statements.

Translation rules: are statements of the form:

p_1 {action$_1$}
p_2 {action$_2$}

p_n {action$_n$}

where each p_i is a regular expression and each action$_i$ is a program fragment describing what action the lexical analyzer should take when pattern p_i matches a lexeme. For example p_i may be an if statement and the corresponding action$_i$ is {return(IF)}.

Lex program specification

How to compile and run the Lex program specification

First use a word processor (for example mule) and create your Lex specification program and then save it under any name but it must have the extension .l (for example *mylexprogram*.l)

Next compile the program using the *UNIX* Lex command which will automatically generate the Lex C program under the name lex.yy.cc

Finally use the UNIX C compiler cc to compile the C program lex.yy.cc

```
% lex mylexprogram.l
% cc lex.yy.c –o first -ll
```

Lex program specification

Example 1   Simple verb recognizer

verb → is | am | was | do | does | has | have

The following is a lex program for the tokens of the grammar

```
%{ /* This is a very simple Lex program for a few verbs recognition */ %}
%%
[ \t]+                                   /* ignore whites pace */
is | am | was | do | does | has | have   {printf(“%s: is a verb\n”, yytext); }
[a-zA-Z]+                                {printf(“%s: is other\n”, yytext); }
%%
main() { yylex() ; }
```

Lex program specification

Example 2   consider the following grammar

statement → if *expression* then *statement*
  | if *expression* then *statement* else *statement*

expression → *term* **relop** *term* | *term*

term → **id** | **number**

With the following regular definitions

letter → [A-Za-z]
digit → [0-9]

if → *if*
then → *then*
else → *else*
relop → < | <= | = | <> | > | >=
id → letter (letter | digit)*

number → digit$^+$ (. digit$^+$)? (E(+|-)? digit+)?

Lex program specification

Example 2

The following is a lex program for the tokens of the grammar

```
%{ /* Here are the definitions of the constants LT, LE, EQ, NE, GT, IF, THEN, ELSE, ID, NUMBER, RELOP */ %}
    delim [ \t\n]
    ws {delim}+
    letter [A-Za-z]
    digit [0-9]
    id {letter}({letter}|{digit})*
    number {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?
  %%
    {whitespace}
    if {return(IF);}
    then {return(THEN);}
    else {return(ELSE);}
    {id} {yylval = install_id(); return(ID);}
    {number} {yylval = install_num(); return(NUMBER);}
    "<" {yylval = LT; return(RELOP);}
    "<=" {yylval = LE; return(RELOP);}
    "=" {yylval = EQ; return(RELOP);}
    "<>" {yylval = NQ; return(RELOP);}
    ">" {yylval = GT; return(RELOP);}
    ">=" {yylval = GE; return(RELOP);}
  %%
    install_id() {}
    install_num() {}
```