COP 3402 Systems Software

Lexical Analysis

Outline

- Lexical analyzer/Lexer
- Regular expressions
- Deterministic and non-deterministic finite automata
- Transition tables
- Lex: lexical-analyzer generator

Lexical Analyzer

The following slides are based on Chapter 2 *Lexical Analysis* of the book *Modern Compiler Implementation in C* by Andrew Appel.

The lexical analyzer takes a stream of characters and produces a stream of lexical tokens; it discards white space and comments between the tokens.

Lexical tokens
A lexical token is a sequence of characters that can be treated as a unit in the grammar of a programming language.

A programming language classifies lexical tokens into a finite set of token types.

Lexical Analyzer

For example, some of the token types of a programming language such as C are:

Token type	Examples
`ID`	`foo n14 last`
`NUM`	`73 0 00 515 082`
`REAL`	`66.1 .5 10. 1e67 5.5e-10`
`IF`	`if`
`COMMA`	`,`
`NOTEQ`	`!=`
`LPAREN`	`(`
`RPAREN`	`)`

Punctuation tokens such as `IF, VOID, RETURN` constructed from alphabetic characters are called reserved words, and in most languages, cannot be used as identifiers.

Lexical Analyzer

Examples of non-tokens are

comment	`/* try again */`
preprocessor directive	`#include <stdio.h>`
preprocessor directive	`#define NUMS 5 , 6`
macro	`NUMS`
blanks, tabs, and newlines	

In languages weak enough to require a macro processor, the preprocessor operates on the source character stream, producing another character stream that is then fed to the lexical analyzer.

Lexical Analyzer

Given a program such as

```
float match0(char *s) /* find a zero */
{if (!strncmp(s, “0.0”, 3))
  return 0.;
}
```

the lexical analyzer will return the stream

```
FLOAT ID(match0) LPAREN CHAR STAR ID(s) RPAREN
LBRACE IF BANG ID(strncmp) LPAREN ID(s) COMMA STRING(0.0) COMMA
NUM(3) RPAREN RPAREN RETURN REAL(0.0) SEMI RBRACE EOF
```

where the token-type of each token is reported.

Some of the tokens, such as identifiers and literals, have **semantic values** attached to them, giving auxiliary information in addition to the token type.

PL/0 Symbols

Example program written in PL/0:

```
const m = 7, n = 85;
var  i, x, y, z, q, r;
procedure mult;
var a, b;
begin
  a := x;  b := y; z := 0;
  while b > 0 do
    begin
       if odd x then z := z+a;
       a := 2*a;
       b := b/2;
    end
end;
begin
  x := m;
  y := n;
  call mult;
end.
```

PL/0 Symbols

Example program written in PL/0:

```
const m = 7, n = 85;
var  i, x, y, z, q, r;
procedure mult;
var a, b;
begin
  a := x;  b := y; z := 0;
  while b > 0 do
    begin
       if odd x then z := z+a;
       a := 2*a;
       b := b/2;
    end
end;
begin
  x := m;
  y := n;
  call mult;
end.
```

Reserved Words (Keywords)

PL/0 Symbols

Example program written in PL/0:

```
const m = 7, n = 85;
var  i, x, y, z, q, r;
procedure mult;
var a, b;
begin
  a := x;  b := y; z := 0;
  while b > 0 do
    begin
       if odd x then z := z + a;
       a := 2 *  a;
       b := b / 2;
    end
end;
begin
  x := m;
  y := n;
  call mult;
end.
```

Operators

+, -, *, /, <, =, >, <=, <>, >=, :=

PL/0 Symbols

Example program written in PL/0:

```
const m = 7, n = 85;
var  i, x, y, z, q, r;
procedure mult;
var a, b;
begin
  a := x;  b := y; z := 0;
  while b > 0 do
    begin
       if odd x then z := z + a;
       a := 2 *  a;
       b := b / 2;
    end
end;
begin
  x := m;
  y := n;
  call mult;
end.
```

Operators

+, -, *, /, <, =, >, <=, <>, >=, :=

Special Symbols

(,), ,, ., ;

PL/0 Symbols

Example program written in PL/0:

```
const m = 7, n = 85;
var  i, x, y, z, q, r;
procedure mult;
var a, b;
begin
  a := x;  b := y; z := 0;
  while b > 0 do
    begin
       if odd x then z := z + a;
       a := 2 *  a;
       b := b / 2;
    end
end;
begin
  x := m;
  y := n;
  call mult;
end.
```

Numbers such as
7, 85, 0, 2, ...

PL/0 Symbols

Example program written in PL/0:

```
const m = 7, n = 85;
var  i, x, y, z, q, r;
procedure mult;
var a, b;
begin
  a := x;  b := y; z := 0;
  while b > 0 do
    begin
       if odd x then z := z + a;
       a := 2 *  a;
       b := b / 2;
    end
end;
begin
  x := m;
  y := n;
  call mult;
end.
```

Identifiers:

- a letter or
- a letter followed by more letters or
- a letter followed by more letters or digits.

Examples:

`x, m, celsius, mult, intel486`

Designing a Lexer

Define identifiers and numbers (tokens with **semantic values**), reserved words, and remaining lexical tokens in PL/0

Identifiers: a lower case letter, followed by sequence consisting of digits or letters (total length 16 or less), not equal to a reserved word

Numbers: integer numbers; max value `2^16-1`

Reserved words:

```
begin call const do end if odd procedure then var while
```

Operators and special symbols:

```
+  -  *  /
(  )
:=
=
<>
,  ;  .
<= >= <  >
```

Designing a Lexer

```
/* PL0 token types */

typedef enum token {
  nulsym = 1, identsym, numbersym, plussym, minussym,
  multsym, slashsym, oddsym, eqsym, neqsym, lessym, leqsym,
  gtrsym, geqsym, lparentsym, rparentsym, commasym, semicolonsym,
  periodsym, becomessym, beginsym, endsym, ifsym, thensym,
  whilesym, dosym, callsym, constsym, varsym, procsym, writesym,
  readsym , elsesym
} token_type;
```

Designing a Lexer

I found it helpful to define the following arrays in my implementation of the lexer.

For instance, to check if a keyword occurs, I go through the keyword array and check if the corresponding substring starts at the current position in the source code file. (But be careful an identifier could be called **var**iable, so you have to do an additional check.)

```
/* names of reserved words */

char *keyword[] = {
  “null”, “begin”, “call”, “const”, “do”, “else”, “end”, “if”,
  “odd”, “procedure”, “read”, “then”, “var”, “while”, “write”
};

/* types of reserved words */

int keyword_type[] = {
    nul, beginsym, callsym, constsym, dosym, elsesym, endsym,
    ifsym, oddsym, procsym, readsym, thensym, varsym, whilesym, writesym
};
```

Regular Expressions

The mathematical notion of **regular expression** is very useful to describe lexical token of a programming language.

A **language** is a set of **strings**.

A string is a finite sequence of **symbols**.

The symbols are themselves taken from a finite **alphabet**.

To specify languages (some of which may be infinite) with finite descriptions, we use the notation of **regular expressions**.

Each regular expression stands for a set of string.

Regular Expressions

Symbol
For each symbol a in the alphabet of the language, the regular expression a denotes the language containing just the string a.

Alternation
A string is in the language of M | N if it is in the language of M or in the language of N.

Concatenation
A string is in the language M · N if it is the concatenation of any two strings α and β such that α is in M and β is in N. Often, we just write M N.

Epsilon
The regular expression ε represents a language whose only string is the empty string.

Repetition
A string is in M* if it is the concatenation of zero or more strings, all of which are in M. M* is called the Kleene closure of M.

Regular Expressions

Using symbols, alternation, concatenation, epsilon, and Kleene closure we can specify the set of ASCII strings corresponding to the lexical tokens of a programming language.

For example:

(0 | 1)* · 0	binary numbers that are multiple of two
b*(abb*)*(a |)	strings of a's and b's with no consecutive a's
(a | b)* aa (a | b)	strings of a's and b's containing consecutive a's

In writing regular expressions, the concatenation operator or the epsilon are often omitted, and it is assumed that the Kleene closure binds tighter than concatenation, and concatenation binds tighter than alternation.

Regular Expressions

Some more abbreviations:

[abcd]	means	(a | b | c | d)
[b-g]	means	[bcdefg]
[b-gM-Qkr]	means	[bcdefgMNOPQkr]
M?	means	(M |)
M+	means	M M*
M{n,m}	means	the language of strings that are concatenations of at least n and at most m strings in the language of M

These extensions are convenient, but none extend the descriptive power of regular expressions.

Finite Automata

Regular expressions are convenient for specifying lexical tokens.

But we need a formalism that can be implemented as a computer program.

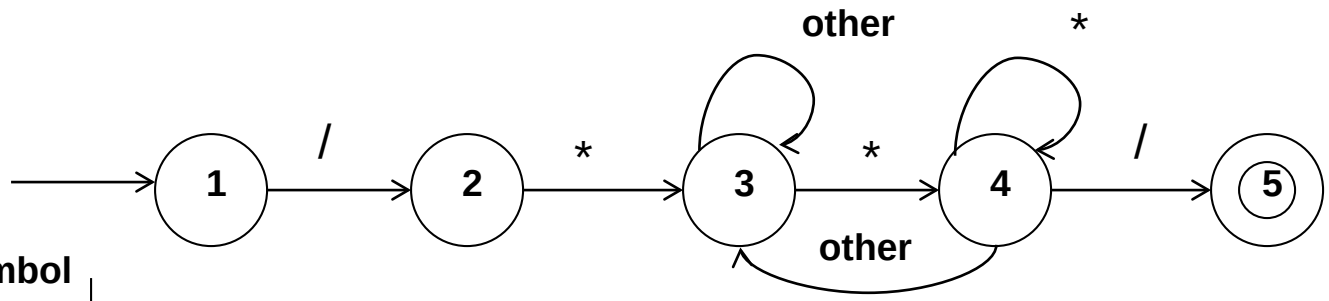
For this we can use **finite automata**.

A finite automaton has a finite set of **states**; **edges** lead from one state to another, and each edge is labeled with a **symbol**.

One state is the **start** state, and certain of the states are distinguished as **final** states.

Example of Finite Automaton

Finite automaton for recognizing C comments.



state \ symbol	/	*	other	final state
1	2			no
2		3		no
3	3	4	3	no
4	5	4	3	no
5				yes

Transition table

Generation of a Lexer

- The regular expressions that describe the lexical tokens of a programming language are combined and give rise to a **non-deterministic** finite automaton.
- The non-deterministic automaton is converted into a **deterministic** finite automaton.
- The deterministic finite automaton is translated into a computer program.

All these steps are handled automatically by the tool **lex**.

Regular expressions

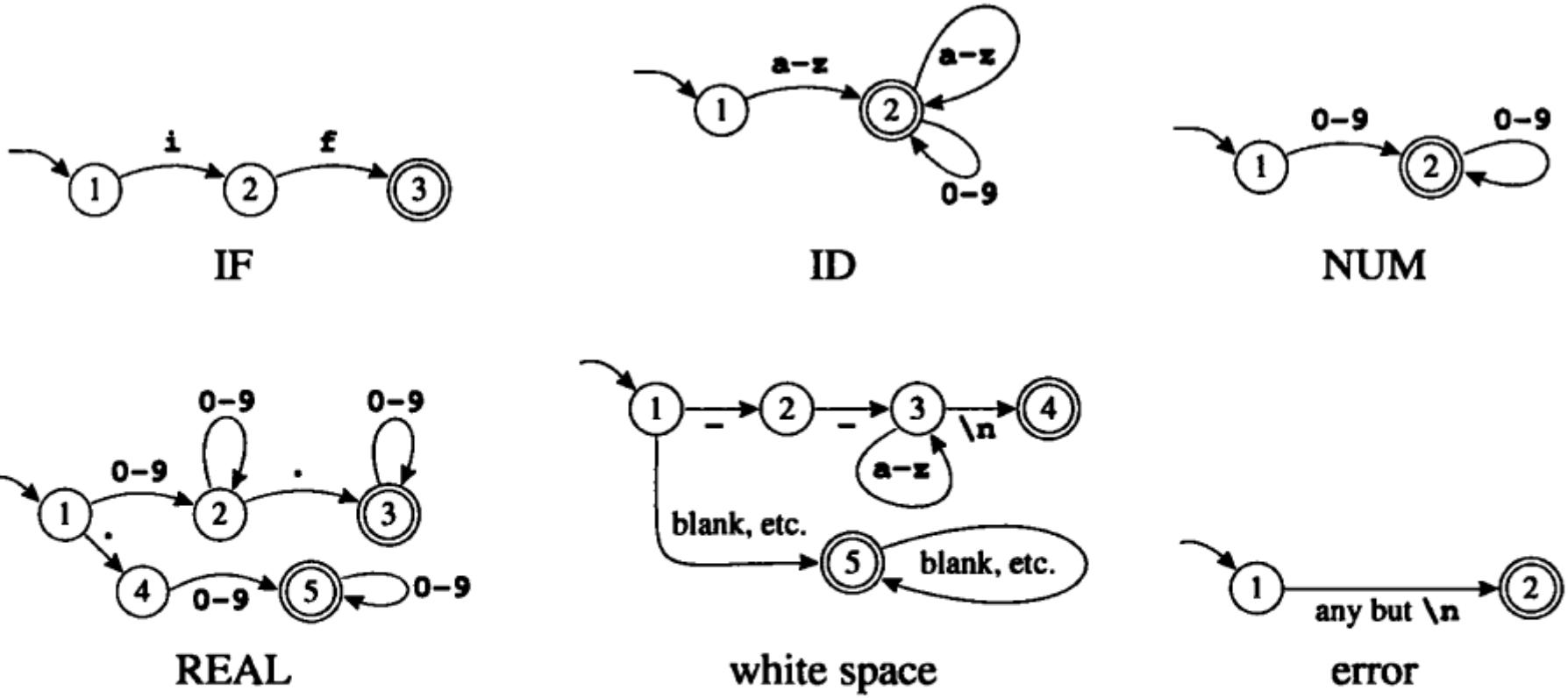


FIGURE 2.3. Finite automata for lexical tokens. The states are indicated by circles; final states are indicated by double circles. The start state has an arrow coming in from nowhere. An edge labeled with several characters is shorthand for many parallel edges.

DFA

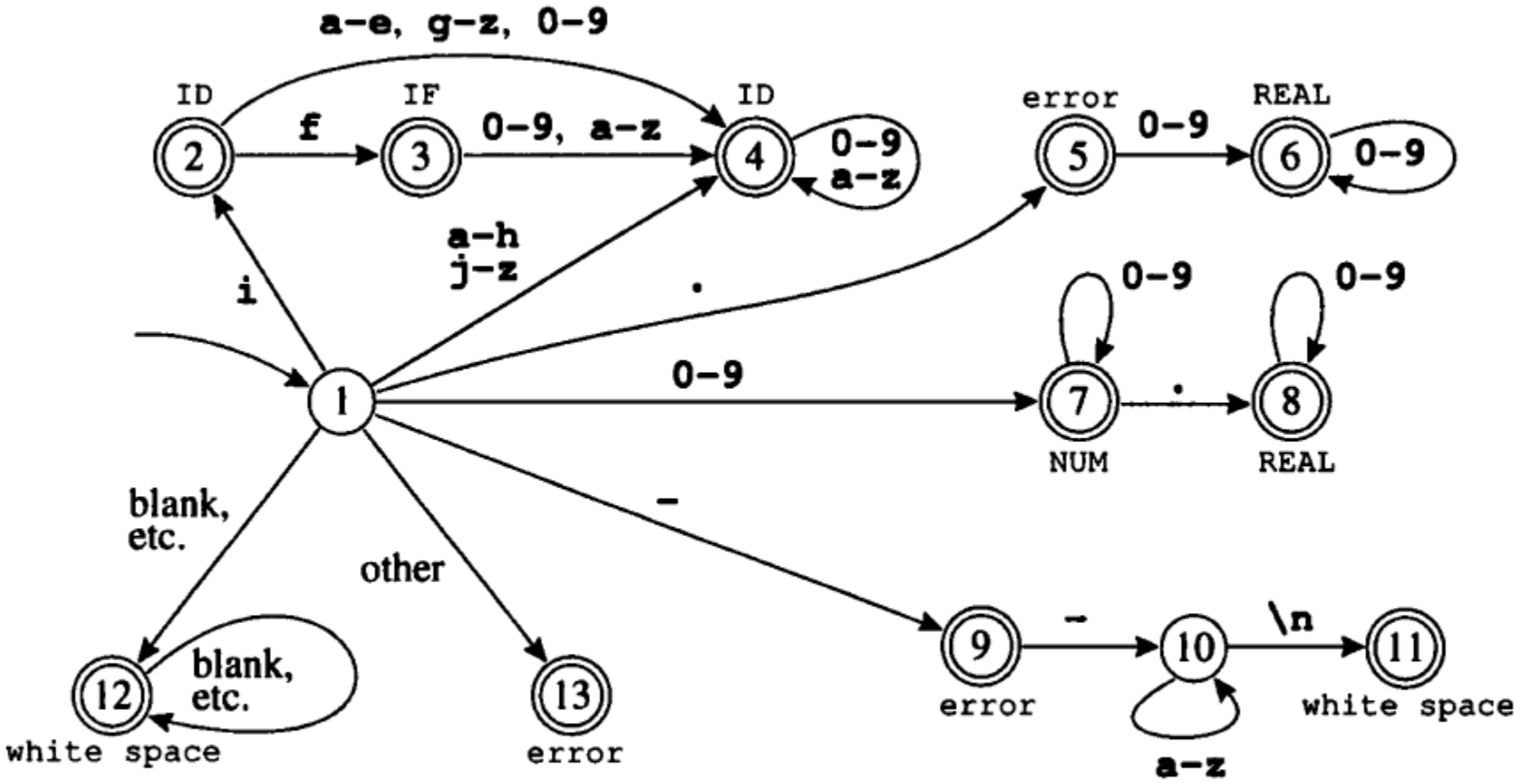


FIGURE 2.4. Combined finite automaton.

Transition matrix

We can encode this machine as a transition matrix: a two-dimensional array (a vector of vectors), subscripted by state number and input character. There will be a "dead" state (state 0) that loops to itself on all characters; we use this to encode the absence of an edge.

```
int edges[][256]={ /* ···0 1 2···-···e f g h i j··· */
/* state 0 */     {0,0,···0,0,0···0···0,0,0,0,0,0···},
/* state 1 */     {0,0,···7,7,7···9···4,4,4,4,2,4···},
/* state 2 */     {0,0,···4,4,4···0···4,3,4,4,4,4···},
/* state 3 */     {0,0,···4,4,4···0···4,4,4,4,4,4···},
/* state 4 */     {0,0,···4,4,4···0···4,4,4,4,4,4···},
/* state 5 */     {0,0,···6,6,6···0···0,0,0,0,0,0···},
/* state 6 */     {0,0,···6,6,6···0···0,0,0,0,0,0···},
/* state 7 */     {0,0,···7,7,7···0···0,0,0,0,0,0···},
/* state 8 */     {0,0,···8,8,8···0···0,0,0,0,0,0···},
   et cetera
}
```

There must also be a "finality" array, mapping state numbers to actions – final state 2 maps to action ID, and so on.

Recognizing the Longest Match

There are two important disambiguation rules used by Lex and other similar lexical-analyzer generators:

Longest match
The longest initial substring of the input that can match any regular expression is taken as the next token.

For instance, `if8` is recognized as `ID` and not `IF` followed by `NUM`.

Rule priority
For a particular longest initial substring, the first regular expression that can match determines the next token.
This means that the order of writing down the regular-expression rules has significance.

Recognizing the Longest Match

It is easy to see how to use the transition table to recognize whether to accept or reject a string.

But the job of a lexical analyzer is to find the longest match, the longest initial substring of the input that is a valid token.

While interpreting transitions, the lexer must keep track of the longest match seen so far, and the position of that match.

Lex

```
%{
/* C declarations */

%}

/* Lex definitions */

%%

/* Regular expressions and actions */
```

Lex

Regular expressions are static and declarative.

Automata are dynamic and imperative.

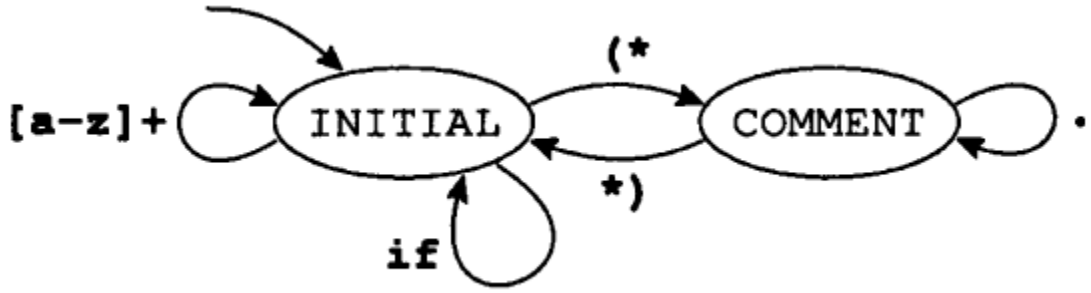
Lex has a mechanism to mix states with regular expressions.

One can declare a set of start states; each regular expression can be prefixed by the set of state in which it is valid.

The action fragments can explicitly change the start state.

In effect we have a finite automaton whose edges are labeled by regular expressions.

Lex



This example shows a language with simple identifiers, if tokens, and comments delimited by (* and *) brackets.

Lex

The Lex specification corresponding to this machine is

```
    ⋮ the usual preamble ...
%Start INITIAL COMMENT
%%
<INITIAL>if      {ADJ; return IF;}
<INITIAL>[a-z]+ {ADJ; yylval.sval=String(yytext); return ID;}
<INITIAL>"(*"    {ADJ; BEGIN COMMENT;}
<INITIAL>.       {ADJ; EM_error("illegal character");}
<COMMENT>"*)"    {ADJ; BEGIN INITIAL;}
<COMMENT>.       {ADJ;}
.                {BEGIN INITIAL; yyless(1);}
```

where INITIAL is the "outside of any comment" state. The last rule is a hack to get Lex into this state. Any regular expression not prefixed by a <STATE> operates in all states; this feature is rarely useful.