

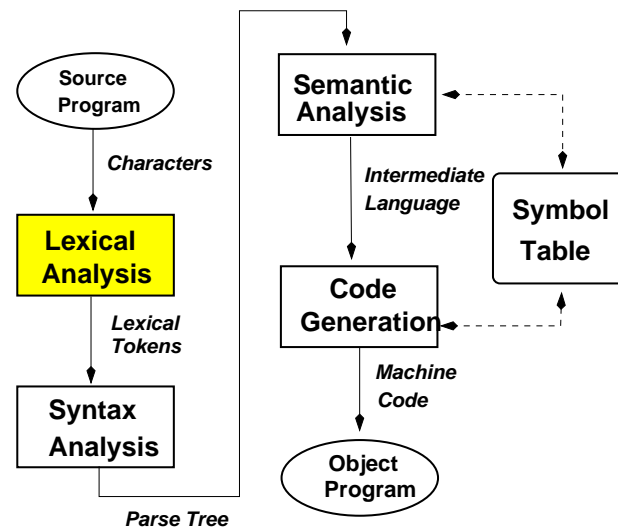
CSC 488S/CSC 2107S Lecture Notes

These lecture notes are provided for the personal use of students taking CSC488H1S or CSC2107HS in the Winter 2015/2016 term at the University of Toronto

Copying for purposes other than this use and all forms of distribution are expressly prohibited.

©David B. Wortman, 2008,2009,2010,2012,2013,2014,2015,2016

©Marsha Chechik, 2005,2006,2007



0

21

Reading Assignment

Fischer, Cytron, LeBlanc

Chapter 3

Omit Section 3.8

22

Lexical Analysis (scanning)

- Lexical analysis transforms its input (a stream of *characters*) from one or more source files into a stream of language-specific *lexical tokens*.
- Lexical Analysis Tasks
 - Delete irrelevant information (whitespace, comments)
 - Determine lexical token boundaries in source stream
 - Transmit lexical tokens to next pass.
 - Deal with ill-formed lexical tokens, recover from lexical errors.
 - Process literal constants (optional).
 - Transmit source coordinates (file, line number) to next pass.
- **Lexical analysis must be really efficient for good compiler performance.**

23

Lexical Analysis

- Typical programming language objects a lexical analyzer must deal with.
 - Identifiers
 - Reserved words or Keywords.
 - Literal constants: integer, real, string ...
 - Special characters: + - * / () & . . .
 - Comments
- A programming language *should* be designed so that the lexical analyzer can determine lexical token boundaries easily.
- In most languages, lexical tokens are designed to be easily separated based on a single left to right scan of the input stream without backup.

24

Running Example Program

```
/* Turing program to find prime numbers <= 10,000 */
const n := 10000
var sieve, primes : array 2 .. n of boolean
var next : int := 2
for k1 := 2 .. n /* initialize */
    sieve[ k1 ] := true    primes[ k1 ] := false
end for
loop
    exit when next > n
    primes[ next ] := true
    for k2 : next .. n by next
        sieve[ k2 ] := false
    end for
    loop /* find next prime */
        exit when next > n or sieve[ next ]
        next += 1
    end loop
end loop
/* prime[i] is true only for prime numbers */
```

25

Lexical Analysis – Running Example Program

Input - a stream of characters

```
/* Turing program to find prime numbers <= 10,000 */
const n := 10000
var sieve, primes : array 2 .. n of boolean
var next : int := 2
for k1 := 2 .. n /* initialize */
    sieve[ k1 ] := true
    primes[ k1 ] := false
end for
loop
    exit when next > n
    primes[ next ] := true
    for k2 : next .. n by next
        sieve[ k2 ] := false
    end for
    loop /* find next prime */
        exit when next > n or sieve[ next ]
        next += 1
    end loop
end loop
/* prime[i] is true only for prime numbers */
```

Output - a stream of lexical tokens

```
const n := 10000
var sieve, primes : array 2 .. n of boolean
var next : int := 2
for k1 := 2 .. n
    sieve [ k1 ] := true
    primes [ k1 ] := false
end for
loop
    exit when next > n
    primes [ next ] := true
    for k2 : next .. n by next
        sieve [ k2 ] := false
    end for
    loop
        exit when next > n or sieve [ next ]
        next += 1
    end loop
end loop
```

26

Building Lexical Analyzers

- Describe the lexical structure of the language using *regular expressions*
- Generate a non-deterministic finite automata (NFA) that recognizes the strings (regular sets) specified by the regular expressions.
- Convert the NFA into a *deterministic finite automata* (DFA).^a
- Implement the DFA using a table-driven algorithm or as hand coded algorithm.

^aSee Fischer, Cytron, LeBlanc Section 3.8 for the formal details

27

Regular Expressions

- Regular expression notation is a convenient way to precisely describe the lexical tokens in a language.
- The sets of strings defined by regular expressions are *regular sets*. Each regular expression defines a regular set.
- Let Σ (*Vocabulary*) be a finite set of characters.
Typically *ASCII* or *Unicode*
 Σ defines the input to the lexical analyzer.
- λ is the empty or null string. \emptyset is the empty set.
- Concatenation is used to build strings from the characters in Σ .
- Characters may be quoted in single quotes ' to avoid ambiguity.
Mandatory for the meta characters: ' (') ' ' | ' ' + ' ' * '

28

Regular Expression Definition

- \emptyset is a regular expression denoting the empty set.
- λ is a regular expression denoting the set that contains only the empty string.
Note $\emptyset \neq \lambda$
- A symbol S (a character in Σ) is a regular expression denoting a set containing only S.
- If A and B are regular expressions then so are:

A B	Alternation
A B	Concatenation
A*	Kleene closure, zero or more copies of A
(A)	Parenthesized regular expression
- Useful Sets:

A^+	Positive closure, One or more strings in A
Not(A)	Complement of A with respect to Σ ($\Sigma - A$)
A^k	k fold Concatenation, exactly k copies of strings from A

29

Regular Expression Examples

Digit	(0 1 2 3 4 5 6 7 8 9)
Letter	(A B C D E F G H I J K L M N O P Q R S T U V W X Y Z)
Hex Digit	(Digit A B C D E F)
Integer	Digit ⁺
OptionalSign	('+' '-' λ)
Signed integer	OptionalSign Integer
Identifier	Letter (Digit Letter '_')*
Real Number	Digit ⁺ '.' Digit* ((E e) OptionalSign Digit ⁺ λ)
Java comment	// (Not (Eol)) * Eol

30

Finite Automata

- Finite automata (state machines) can be used to recognize tokens specified by a regular expression.
- A finite automaton (FA) is
 - A finite set of states
 - A set of transitions from one state to another.
Transitions depend on input characters from the vocabulary Σ
 - A distinguished *start* state.
 - A set of final (*accepting*) states.
- Finite automata can be represented graphically using state transition diagrams.
- If the FA has a *unique* transition for every (state, input character) pair then it is a *deterministic* FA (DFA).
Being deterministic is good.

31

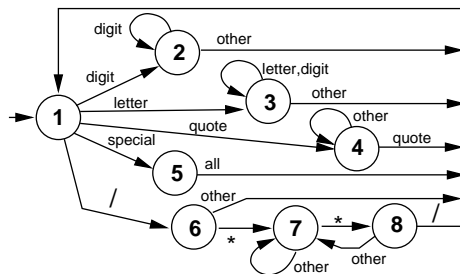
A typical lexical analyzer built using a DFA works like this

- The input to the DFA is the next input character to the lexical analyzer.
- The main state of the DFA uses the character to classify the type of possible token.
e.g. identifier, number, special character, comment, whitespace.
- For lexical tokens that can consist of more than one character, the main state transitions to a sub-state that deals with accumulating one type of lexical token.
e.g identifier, number
- Once a complete token has been recognized by the DFA, it is packaged into a lexical token data structure and sent onward to semantic analysis.
- Each sub-state consumes the characters in its lexical token and ultimately returns to the main state to deal with the character immediately after its token

See Slides 33 and 34 for examples of DFA based table driven lexical analyzers.

32

State	Input Character							Purpose
	blank	digit	letter	quote	Special	/	*	
1	+1	2	3	+4	5	+6	5	blanks & dispatch
2	1	+2	1	1	1	1	1	integers
3	1	+3	+3	1	1	1	1	identifiers
4	+4	+4	+4	+1	+4	+4	+4	'string'
5	1	1	1	1	+1	1	+1	special chars.
6	1	1	1	1	1	1	+7	emit token '/'
7	+7	+7	+7	+7	+7	+7	+8	comment body
8	+7	+7	+7	+7	+7	+1	7	end comment



34

Example – Table Driven DFA Scanner

- Example of implementing a simple DFA scanner using a small table to define the DFA.
- Language with whitespace, integers, identifiers, strings, special characters, C style comments.
- Make lexical analysis decisions with 1 character look ahead.
- Advance in input, save start of long tokens.
- Save input pointer on entry to each state.
If appropriate emit accumulated token on return to state 1
- Notation:
N - change to state N
+N - advance input pointer and change to state N

33

Example – Hand Coded DFA Lexical Analyzer

```
while( ! eof ) {
    switch( currentChar ) {
        case letter:      accumulate identifier
                          distinguish identifiers
                          or reserved word
        case digit:      accumulate number
                          distinguish integer, float
        case quote:      accumulate string
        case comtStart:  discard comment
        case special:    accumulate special token
        case whiteSpace: discard white space
    }
    if( tokenProduced )
        emit token
}
```

35

Example - Real Constants in Turing

- **Reference Manual definition - literal real constant:**
 - An *explicitRealConstant* consists of an optional plus or minus sign a significant digits part and an exponent part.
 - The significant digits part ... consists of a sequence of one or more digits optionally containing a decimal point. The decimal point is allowed to follow the last digit as in 16. or precede the first digit as in .25
 - The exponent part consists of the letter e or E followed optionally by a plus or minus sign followed by one or more digits. If the significant figures [sic] part contains a decimal point then the exponent part is not required.
- **Issues**
 - Must have at least one digit in the significant digits part. Must have at least one digit in the exponent part.
 - Context sensitive to distinguish integer constants from real constants.
 - Context sensitive: exponent can be omitted iff there was a decimal point in the significant digits part.
 - Leading/trailing decimal point interacts with use of .. for integer subrange. e.g. 3.4 vs. 3..4
 - Handle leading sign as unary + or - during syntax analysis. Lexical analyzer can't easily distinguish $X = 3.5$ from $X = -3.5$ or $X = - -3.5$.
 - Must detect malformed constants like 3.4.5E+
 - Arbitrary number of digits in exponent part is unrealistic. Exponent range is -38 .. +38 on 32-bit machines. But must allow 1.0E+000000000000000000003

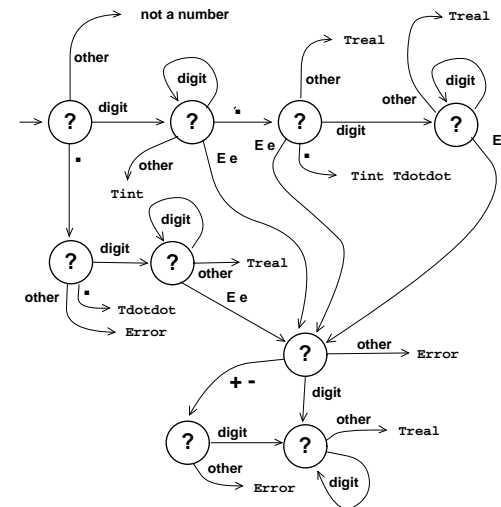
Building Tables During Lexical Analysis

- To save space in the lexical token stream, the lexical analyzer can build tables of certain kinds of tokens, e.g. identifiers, string constants, other constants. These tables are then used by later passes
- The lexical token that gets passed on to the syntax analysis pass contains a field that identifies the type of token and the tokens index into an appropriate table.
- **Indices that uniquely identify tokens are sufficient for most purposes in syntax and semantic analysis.**
- Examples:

Original Token	Encoded as		
veryLongIdentifier	<table border="1" style="display: inline-table;"><tr><td>Ident</td><td>23</td></tr></table>	Ident	23
Ident	23		
shorterIdentifier	<table border="1" style="display: inline-table;"><tr><td>Ident</td><td>29</td></tr></table>	Ident	29
Ident	29		
"Hello World"	<table border="1" style="display: inline-table;"><tr><td>StrCon</td><td>19</td></tr></table>	StrCon	19
StrCon	19		
- If the compilers symbol table is based on a hash table, could also compute (once) the hash key for each identifier during lexical analysis.

Original Token	Encoded as	
veryLongIdentifier	Ident	23
shorterIdentifier	Ident	29
"Hello World"	StrCon	19

DFA Recognizer for Turing Numeric Constants



Handling Reserved Words and Keywords

- In many programming languages, reserved words or keywords are lexically indistinguishable from identifiers.
- There are several possible approaches for recognizing reserved words:
 - Write a regular expression to recognize each special word.
Order the regular expressions to recognize these words before identifiers.
This is **SLOW**. It requires buffering and backup in the lexical analyzer.
 - Write a regular expression to recognize all identifiers.
Use a fast table lookup to check whether the identifier is really a reserved word.
Production compilers typically use this approach.
 - If the lexical analyzer is building an identifier table, preload the reserved words into the table so that low table indices indicate special words
- For keyword languages, flag the identifier as a *maybe keyword* and let syntax analysis decide.

What Makes Lexical Analysis Difficult

- Dealing with long tokens, e.g. strings in PL/I can contain 32,768 characters.
- Tokens that can be split across input boundaries.
- Special handling in character strings. e.g. \nnn in C.
- Context sensitive input, e.g.string constant concatenation in C. .e.g "AB" "CD"
- Processing numeric constant values close to the hardware limits for minimum and maximum values e.g. -2147483648 or 1.7976931348623157e+308
This requires some **very careful computation** in the part of the lexical analyzer that converts constants.
- Separation requiring lookahead.
More than one character lookahead is usually painful.
e.g. 123456 vs 123456E+12
- Dealing with large initializations, e.g. initialization in array declarations in C and Java

40

Lexically Challenged Languages – FORTRAN

- Designed before lexical analysis and parsing were well understood
 - Keywords not reserved words
 - Blank insensitive
- Example: DO 10 I = 1

42

Lexical Analysis - Good Language Design

- Different lexical token classes begin with distinct characters.
identifiers (letters), numbers (digits), strings (")
- Reserved words, not keywords.
- No token suffixes that change token meaning.
- Reasonable limits of maximum size of constants.
- Avoid complex structure in constants.

Many modern programming languages satisfy these constraints.

41

Lexically Challenged Languages – PL/I

- Designed to be all things to all programmers
 - Keywords not reserved words
 - Awkward lexical structure
- IF IF = THEN THEN THEN = ELSE ELSE ELSE = IF = THEN
- '101011011101001110110110110001'

43

Managing Source Program Coordinates

- The subsequent phases of the compiler need to know where each lexical token originated to be able to emit source code specific error messages. e.g.,
`Syntax Error on line 23 of file foo.c`
- Conceptually tag each lexical token with an identification of the source file and line number where it originated. Possibly also include the offset in the line. See 53
- More practical systems are based on transmitting only *changes* in source coordinates (as invisible internal lexical tokens) to the subsequent phases. Each phase must then include logic to keep track of the current source program coordinate.

48

Lexical Analyzer Optimization

- First design a *correct* lexical analyzer.
Then optimize it for speed.
- Most important optimizations^a :
 - Input in bulk, not character by character.
 - Touch each input character as few times as possible.
 - Don't copy input if you can avoid it.
 - Discard white space and comments efficiently.
Often 50% .. 60% of the source.
 - Make reserved word testing as fast as possible.
 - Evaluate literal constants efficiently.
- Optimizations should not make lexical analyzer difficult to maintain or modify.

^aSee: W. M. Waite, The Cost of Lexical Analysis, Software—Practice and Experience, v.16, n.5, May 1986

50

Source Coordinate Management Example

- Send a *new source file token* at
 - The start of compilation.
 - The start of processing an included file (e.g. `#include`)
The lexical analyzer may get this information from a pre processor.
 - At the end of processing an included file (e.g. revert to previous file)
- Send a *new source line token* at
 - The start of compilation.
 - Each time the lexical analyzer moves to a new line containing a lexical token (optimizes whitespace deletion).
 - At the start of processing an included file.
 - At the end of processing an included file to revert to the previous files coordinate system.

49

Example - Accumulating Identifiers

- Naive

```
char ch, idBuffer[200] ;
int i ;
. . .
i = 0 ;
while( ('a' <= ch && ch <= 'z') || ( 'A' <= ch && ch <= 'Z' )
      || ( '0' <= ch && ch <= '9' ) || ch == '_' ) {
    idBuffer[i++] = ch ;
    ch = getchar() ;
}
/* identifier is in idBuffer[ 0 .. i - 1 ] */
```

- Optimized

```
BOOL identifierChar[ 256 ] ; /* true for a..z, A..Z, 0..9 _ */
/* false otherwise */
char * iStart, * iEnd, *inPtr ;
. . .
iStart = inPtr ;
while( identifierChar[ *inPtr++ ] ) ;
iEnd = inPtr ;
/* Identifier is in input buffer iStart .. iEnd - 1 */
```

51

Scanner Generators - Lex, Flex & JFlex

- Lex, Flex (Flexible Lex) and JFlex (Flex for Java) are automated *scanner generators* that transform a regular expression description of a language's lexical tokens into source code for a scanner.
- The generated scanners are good for student projects and prototype compilers but may not be efficient enough for production compilers.
- A regular expression defines a class of lexical token.
The compiler writer can specify an arbitrary piece of code that gets executed when the token is recognized.

```
ident  = [A-Za-z][A-Za-z0-9_]*  
digit  = [0-9]
```

```
{ident}      { return symbol(sym.IDENT, yytext ()); }  
  
{digit}+    { return symbol(sym.INTCONST,  
                          Integer.valueOf (yytext())); }
```

52

JFlex Lexical Token

A *lexical token* is a data structure that is used to pass information from lexical analysis to syntax analysis.

This is the lexical token generated by JFlex in the course project.

```
public class Symbol {  
    int left           // parser data  
                        // source file line number of token  
    int parseStart     // record parse state for this token  
                        // used in the parser stack  
    int right          // parser data  
                        // source file column number of token  
    int sym             // symbol number of terminal or nonterminal  
                        // this is the real lexical token  
    java.lang.Object value // value of lexical token (if any)  
                        // e.g. number, string, identifier  
}
```

53