COMPILER DESIGN

Dr. V. Padmavathi
Associate Professor
Dept. of CSE
CBIT

Text Books:

1. Alfred V Aho, Monica S Lam, Ravi Sethi, Jeffrey D Ullman, “Compilers: Principles Techniques & Tools”, Pearson Education 2nd Edition, 2013.
2. Steven Muchnik, “Advanced Compiler Design and Implementation”, Kauffman, 1998.

Suggested Reading:

1. Kenneth C Louden, “Compiler Construction: Principles and Practice”, Cengage Learning, 2005.
2. Keith D Cooper & Linda Tarezon, “Engineering a Compiler”, Morgan Kafman, Second edition, 2004.
3. John R Levine, Tony Mason, Doug Brown “Lex &Yacc”, 3rd Edition Shroff Publisher, 2007.

Online Resources:

1. http://www.nptel.ac.in/courses/106108052
2. https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/
3. http://en.wikibooks.org/wiki/Compiler_Construction
4. http://dinosaur.compilertools.net/
5. http://epaperpress.com/lexandyacc/

- **Pre-requisites:** Formal Language and Automata Theory, Data Structures.

- **Course Objectives:** The objectives of this course are,

  1. Understand and list the different stages in the process of compilation.

  2. Identify different methods of lexical analysis and design top-down and bottom-up parsers.

  3. Implement syntax directed translation schemes and develop algorithms to generate code for a target machine and advance topics of compilers.

Course Outcomes: On successful completion of the course, students will be able to,

1. Identify the concepts related to translator, tokens, bootstrapping, porting and phases of the compiler.

2. Use grammar specifications and implement lexical analyzer by the help of compiler tools.

3. Explore the techniques of Top down, Bottom up Parsers and apply parsing methods for various grammars.

4. Implement syntax directed translation schemes and relate Symbol Table organization.

5. Analyze the concepts involved in Intermediate, Code Generation and Code Optimization process and understand error recovery strategies and advance topics in compilers.

UNIT - I

- **Introduction to compilers** – Analysis of the source program, Phases of a compiler, grouping of phases, compiler writing tools, bootstrapping, data structures.

- **Lexical Analysis:** The role of Lexical Analyzer, Input Buffering, Specification of Tokens using Regular Expressions, Review of Finite Automata, Recognition of Tokens, scanner generator(lex, flex).

Overview and Logical phases of compiler

The compiler implementation process is divided into two parts:

- **Analysis of a source program**
- **Synthesis of a target program**

Analysis involves analyzing the different constructs of the program

Analysis consists of three phases:

- **Lexical analysis (Linear or scanning)**
- **Syntax Analysis (hierarchical)**
- **Semantic analysis**

Synthesis of a target program includes three phases:

- **Intermediate code generator**
- **code optimizer**
- **code generator**

Terms used in lexical analysis

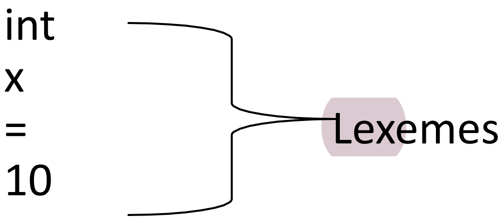
Lexeme: Lexemes are the smallest logical units of a program. It is
the sequence of characters in the source program for which a token
is produced.

Tokens: Class of similar lexemes are identified by the same token.

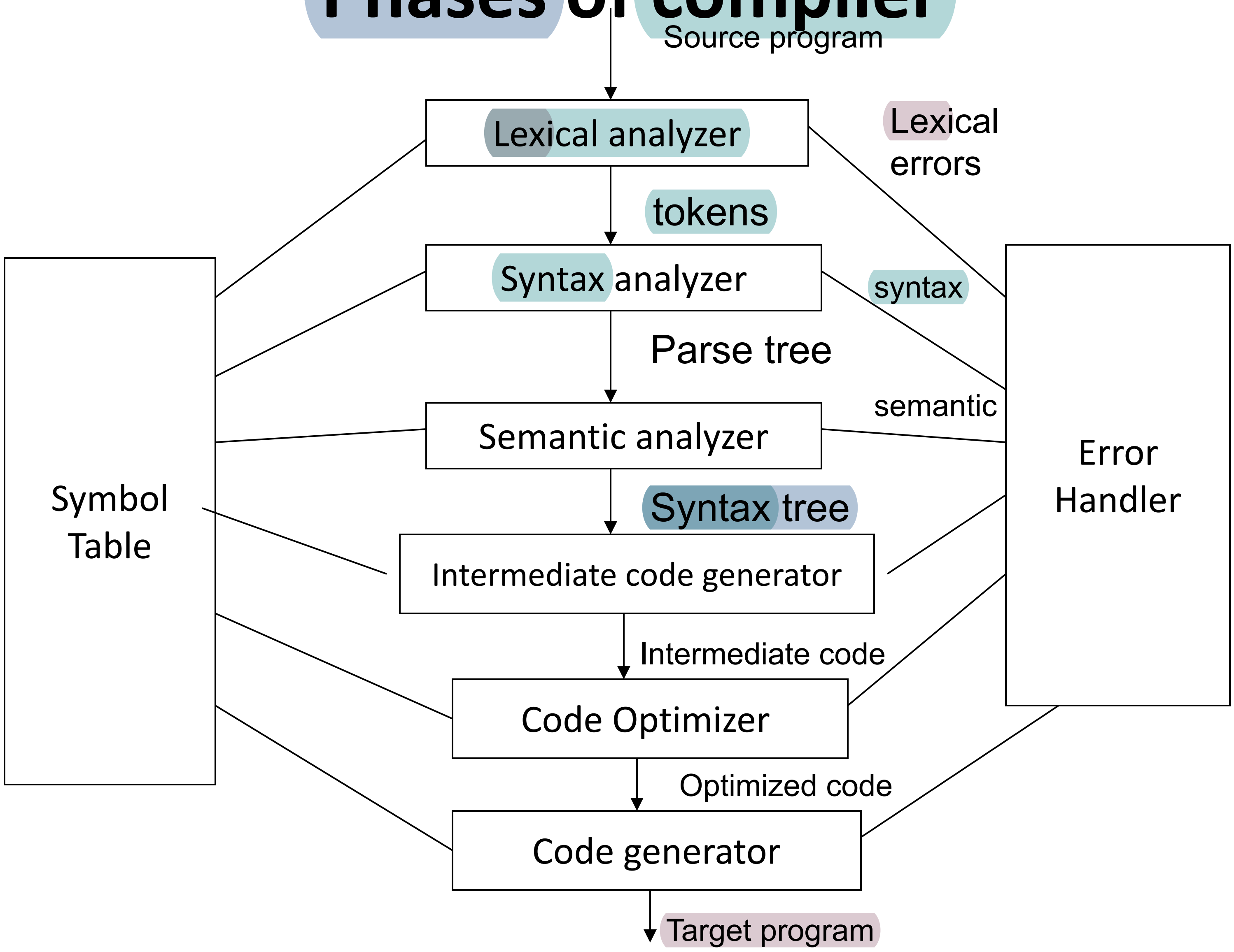
Pattern : Pattern is a rule which describes a token

Ex: Pattern of an identifier is that it should consists of letters and digits but the first character should be a letter.

int x = 10;
int is a lexeme for the token **keyword**
x is a lexeme for the token **identifier**


int
x
=
10
Lexemes

Phases of compiler



Source program

Lexical analyzer

tokens

Syntax analyzer

Parse tree

Semantic analyzer

Syntax tree

Intermediate code generator

Intermediate code

Code Optimizer

Optimized code

Code generator

Target program

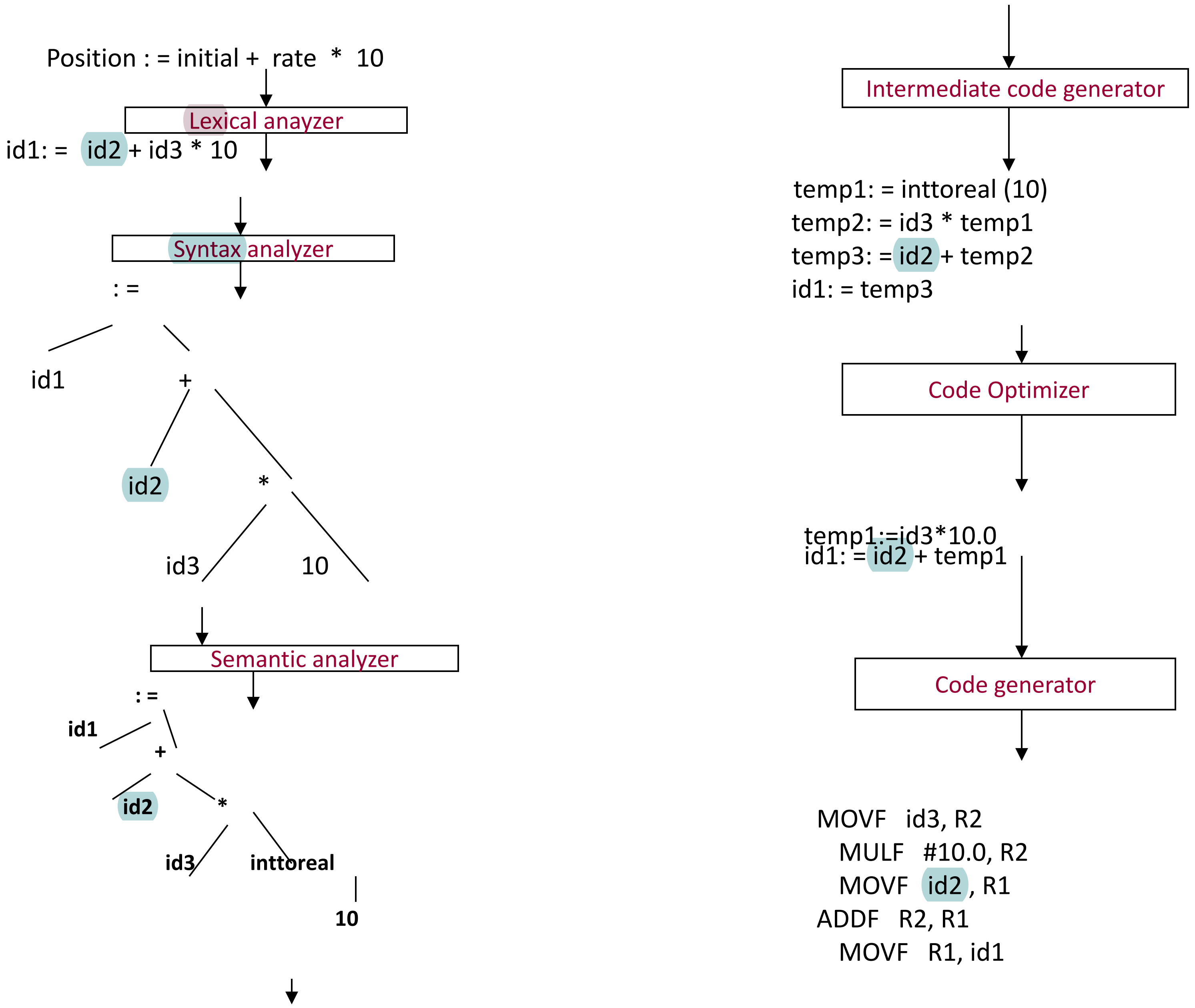
Symbol Table

Error Handler

Lexical errors

syntax

semantic



Position : = initial + rate * 10

Lexical anayzer

id1: = id2 + id3 * 10

Syntax analyzer

: =
id1
+
id2
*
id3
10

Semantic analyzer

: =
id1
+
id2
*
id3
inttoreal
10

Intermediate code generator

```
temp1: = inttoreal (10)
temp2: = id3 * temp1
temp3: = id2 + temp2
id1: = temp3
```

Code Optimizer

```
temp1:=id3*10.0
id1: = id2 + temp1
```

Code generator

```
MOVF   id3, R2
   MULF   #10.0, R2
   MOVF   id2 , R1
ADDF   R2, R1
   MOVF   R1, id1
```

Phases and Passes

Pass

1) Pass is a physical scan over a source program. The portions of one or more phases are combined into a module called pass.

2) Requires an intermediate file between two passes

3) Splitting into more no. of passes reduces memory.

4) Single pass compiler is faster than two pass .

Phase

1) A phase is a logically cohesive operation that takes i/p in one form and produces o/p in another form.

2) No need of any intermediate files in between phases.

3) Splitting into more no. of phases reduces the complexity of the program.

4) Reduction in no. of phases, increases the execution speed

- The repetitions to process the entire source program before generating code are referred as passes.

– A compiler can be one pass or multiple passes

- ➢ One pass: efficient compilation; less efficient target code

  -> space consuming

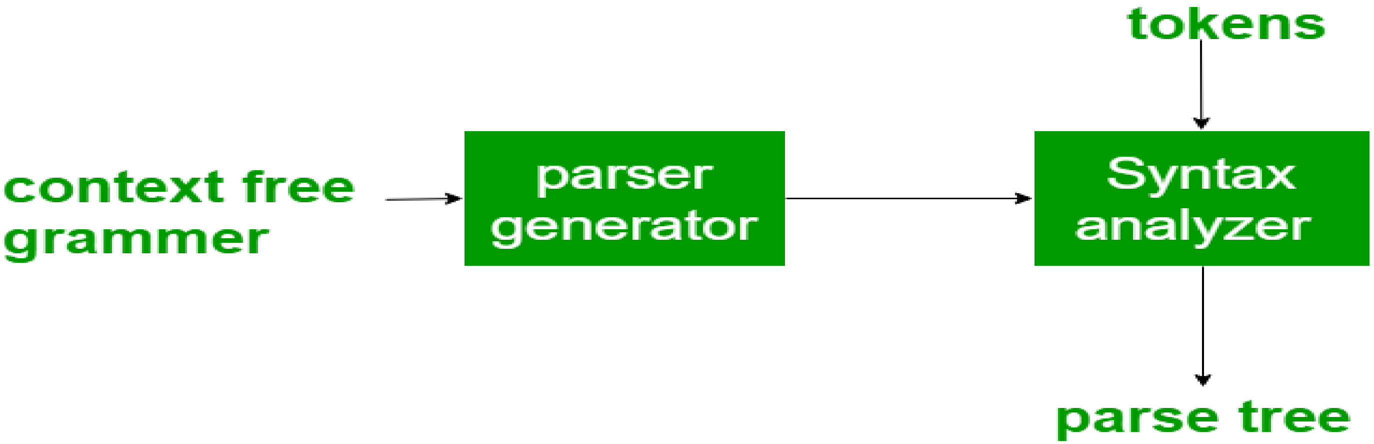
- ➢ Multiple passes: efficient target code; complicated compilation

  -> time consuming

- Most compilers with optimization use more than one pass
- ➢ One Pass for scanning and parsing
- ➢ One Pass for semantic analysis and source-level optimization
- ➢ The third Pass for code generation and target-level optimization

Compiler Writing Tools

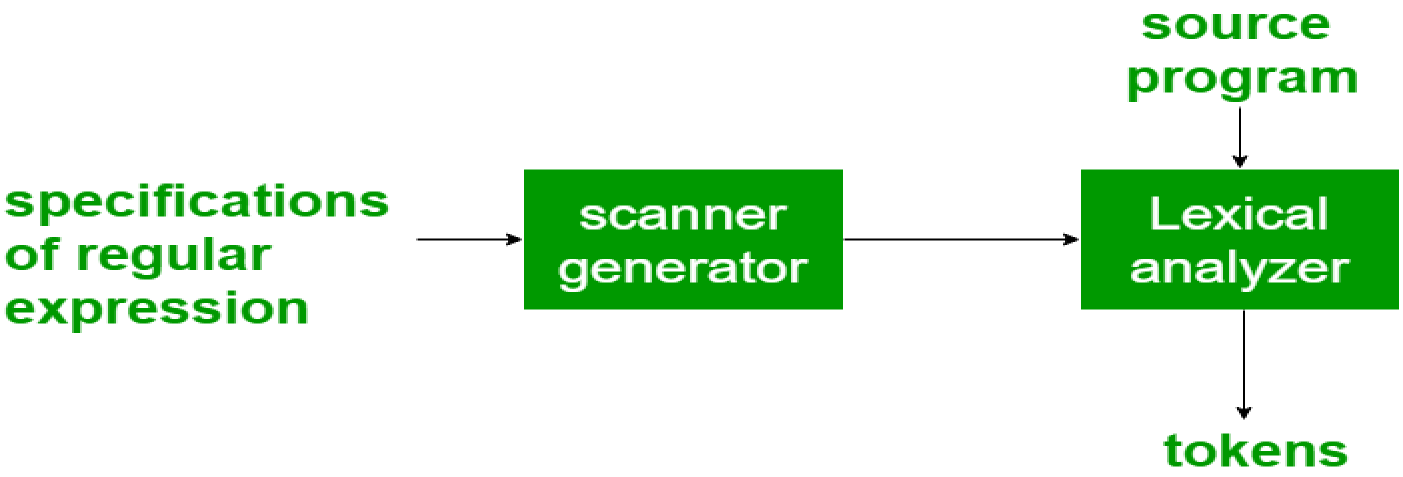
- The compiler writer can use some specialized tools that help in implementing various phases of a compiler. These tools assist in the creation of an entire compiler or its parts.
- Some commonly used compiler construction tools include:
- **Parser Generator –**
  It produces syntax analyzers (parsers) from the input that is based on a grammatical description of programming language or on a context-free grammar. It is useful as the syntax analysis phase is highly complex and consumes more manual and compilation time.



- **Scanner Generator –**

  It generates lexical analyzers from the input that consists of regular expression description based on tokens of a language. It generates a finite automaton to recognize the regular expression.

  Example: Lex



- **Syntax directed translation engines –**
  It generates intermediate code with three address format from the input that consists of a parse tree. These engines have routines to traverse the parse tree and then produces the intermediate code. In this, each node of the parse tree is associated with one or more translations.

- **Automatic code generators –**
  It generates the machine language for a target machine. Each operation of the intermediate language is translated using a collection of rules and then is taken as an input by the code generator. A template matching process is used. An intermediate language statement is replaced by its equivalent machine language statement using templates.

- **Data-flow analysis engines –**
  It is used in code optimization. Data flow analysis is a key part of the code optimization that gathers the information, that is the values that flow from one part of a program to another.
- **Compiler construction toolkits –**
  It provides an integrated set of routines that aids in building compiler components or in the construction of various phases of compile

Other uses Scanning and Parsing Techniques

- Assembler implementation
- Online text searching (GREP, AWK) and word processing
- Website filtering
- Command language interpreters
- Scripting language interpretation (Unix shell, Perl, Python)
- XML Parsing and documentation tree construction
- Database query interpreters

Other uses of Program Analysis Techniques

- Converting sequential loop to a parallel loop
- Program analysis to determine if programs are data-race free
- Profiling programs to determine busy regions
- Program slicing
- Data-flow analysis approach to software testing
  - Uncovering errors along all the paths
  - Dereferencing null pointers
  - Buffer overflows and memory leaks
- Worst case execution time (WCET) estimation and energy analysis

Applications of Compiler technology

Compilers are everywhere…

- Parsers for HTML in web browsers
- Interpreters for JavaScript/ Flash
- Machine code generation of high level languages
- Software testing
- Program optimization
- Malicious code detection
- Design of new computer architectures

  - Compiler in-the-loop hardware development
- Hardware synthesis: VHDL to RTL translation
- Compiled simulation

  - Used to simulate designs written in VHDL

  - No interpretation of design, hence faster

Practical Application-Nature of Compiler Algorithms

- Draws results from mathematical logic, lattice theory, linear algebra, probability, etc.

  - type checking, static analysis, dependence analysis, loop parallelization, cache analysis, etc.
- Greedy algorithms- register allocation
- Heuristic search- list scheduling
- Graph algorithms- dead code elimination, register allocation
- Dynamic programming- instruction selection
- Optimization techniques- instruction scheduling
- Finite automata-lexical analysis
- Pushdown automata- parsing
- Fixed point algorithms- data-flow analysis
- Complex data structures- symbol tables, parse trees, data dependence graphs
- Computer architecture- machine code generation

Bootstrapping

- A technique which rely on partial/inefficient compiler version to create a full/better version.
- often compiling a translator expressed in its own language.

- A process by which a simple language is used to translate more complicated language which in turn translates even more complicated language (and this process continues) is known as bootstrapping.

Full Bootstrapping

- A full bootstrap is necessary when we are building a new compiler from scratch.

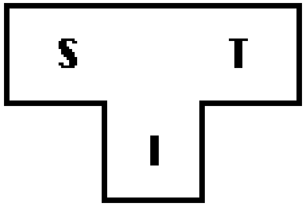
- **Example:**
- We want to implement an Ada compiler for machine *M*. We don't currently have access to any Ada compiler (not on *M*, nor on any other machine).
- Idea: Ada is very large, we will implement the compiler in a subset of Ada and bootstrap it from a subset of Ada compiler in another language. (e.g. C)
- The process illustrated by the **T-diagrams** is called *bootstrapping.*

- A compiler is characterized by

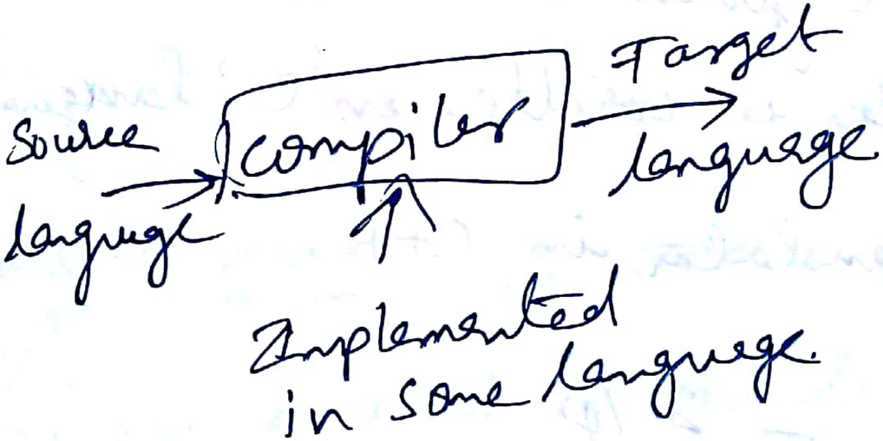
1) source language

2) Target language

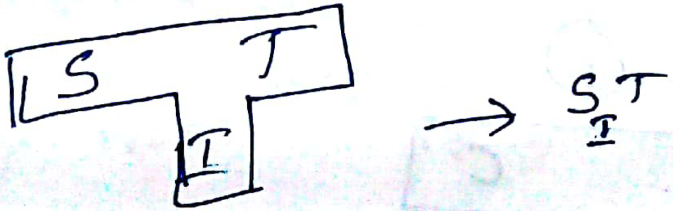
3) Implementation language

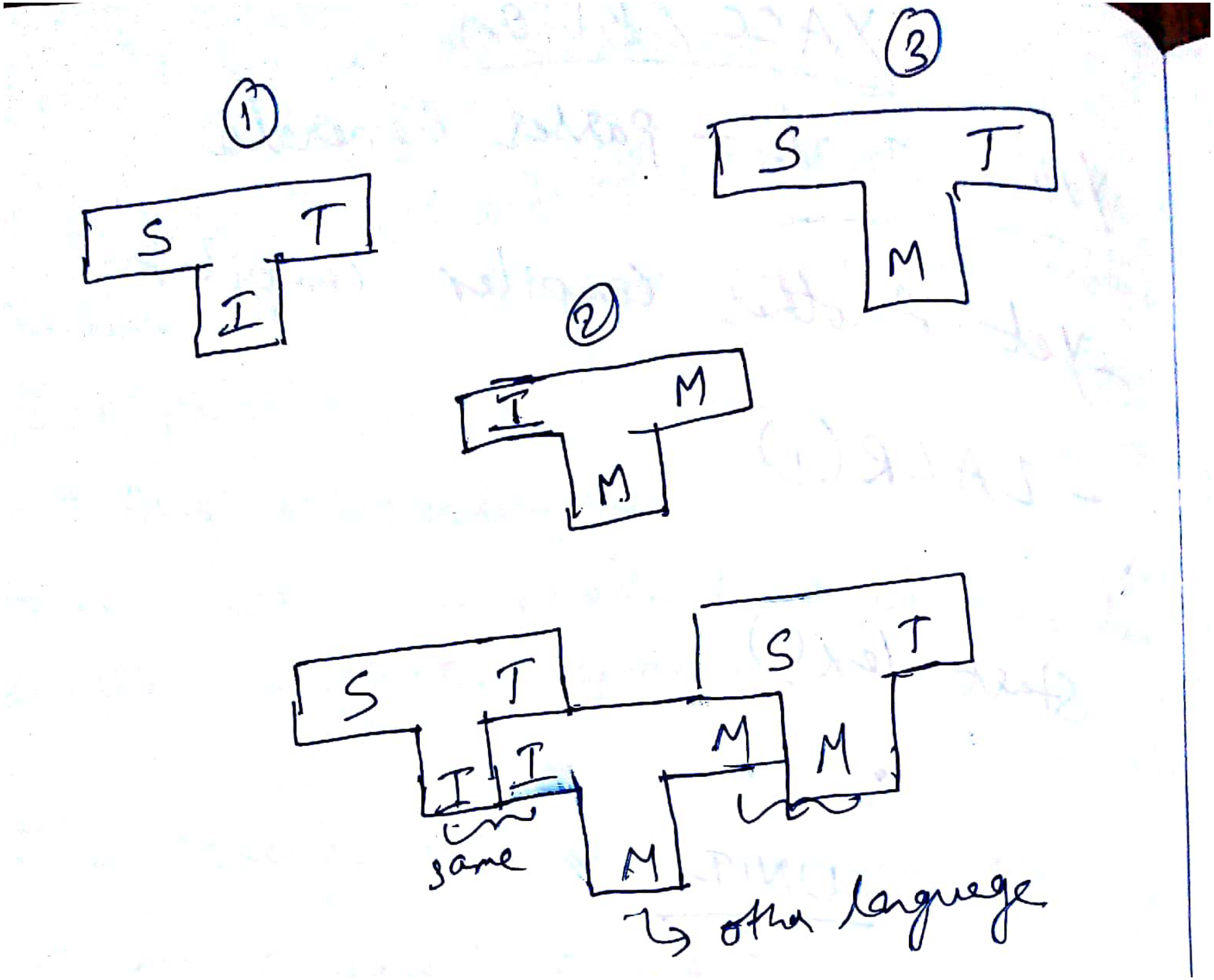


Bootstrapping



T- Diagram




1
S
T
I
2
I
M
M
3
S
T
M
S
T
I
I
M
S
T
M
M
same
M
other language

→ Consider the given scenario, where the pascal ~~parser's~~ translator is written in 'C' language.

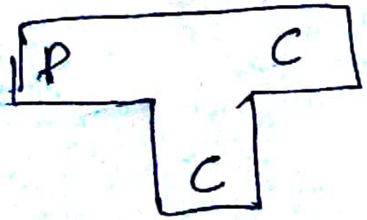
Create pascal translator in C++

pascal code – I/p

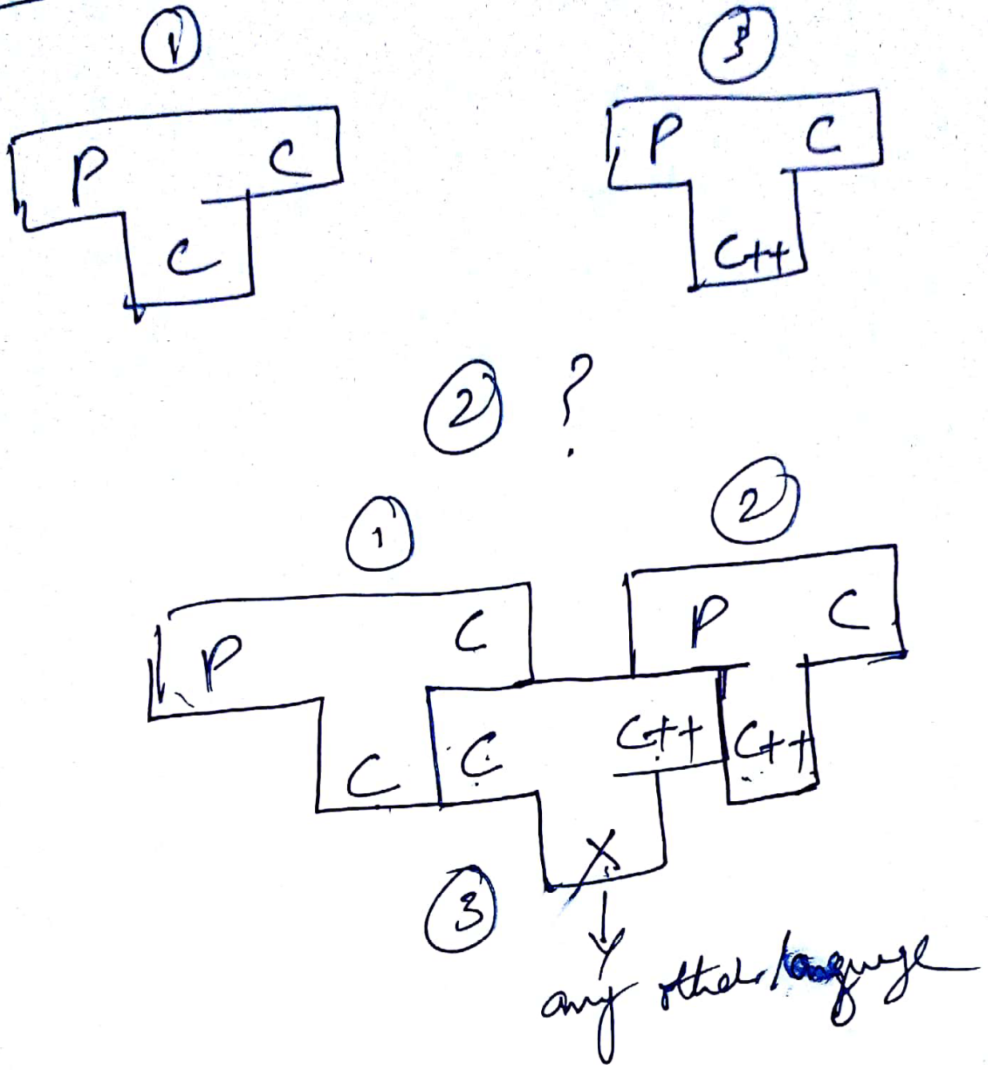
C – O/p

T-Diagram

1



conversion



Data Structures

- The interaction between the algorithms used by the phases of a compiler and the data structures that support these phases, of course is a strong one.
- A compiler should be compiling a program in time proportional to the size of the program.

  Time α Size

  that is O(n) where n is measure of program size (usually number of characters)

The following data structures used by the phases of compiler

- Tokens
- Syntax tree
- Symbol table
- Literal table
- Intermediate Code
- Temporary Files

Tokens

- Scanner converts characters to tokens
- Represents tokens as a value of enumerated datatype
- Preserves the strings of characters
- Other information derived from it like name associated with an identifier or the value of a number.
- Generates one token at a time- single symbol lookahead
- Single global variable used to the token information
- An array of tokens may be used.

Syntax tree

- Standard pointer based structure that is dynamically allocated as parsing proceeds.
- Entire tree can be kept as a single variable pointing to the root.
- Each structure is a record collects information from parser and later phases.
- For example: datatype of an expression may be kept as a field in syntax tree.
- Each node in syntax tree may require different attributes to be stored.

Literal table

- Quick insertion and lookup
- Stores constants and strings used in a program.
- Does not allow deletions since its data applies globally to the program and a constant or string will appear only once in this table.
- Is important in reducing the size by allowing reuse of constants and strings.
- It is needed by the code generator to construct symbolic addresses for literals and entering data definitions in the target code files.

Intermediate code

- Depending on the kind of intermediate code and the kinds of optimizatioms the code may be kept as an array of text strings, temporary text files or as a linked list of structures.

Temporary Files

Symbol table

- Keeps the information associated with identifiers: functions, variables, constants and datatypes.
- Interacts with every phase.
- Since the symbol table accessed so frequently, insertion, deletion and access operations need to be efficient, preferably constant time operations.
- Hence, hash table and other tree structures may be used.
- Sometimes several tables are used and maintained in a list of a stack.

Lexical Analysis

Role of Lexical Analyzer

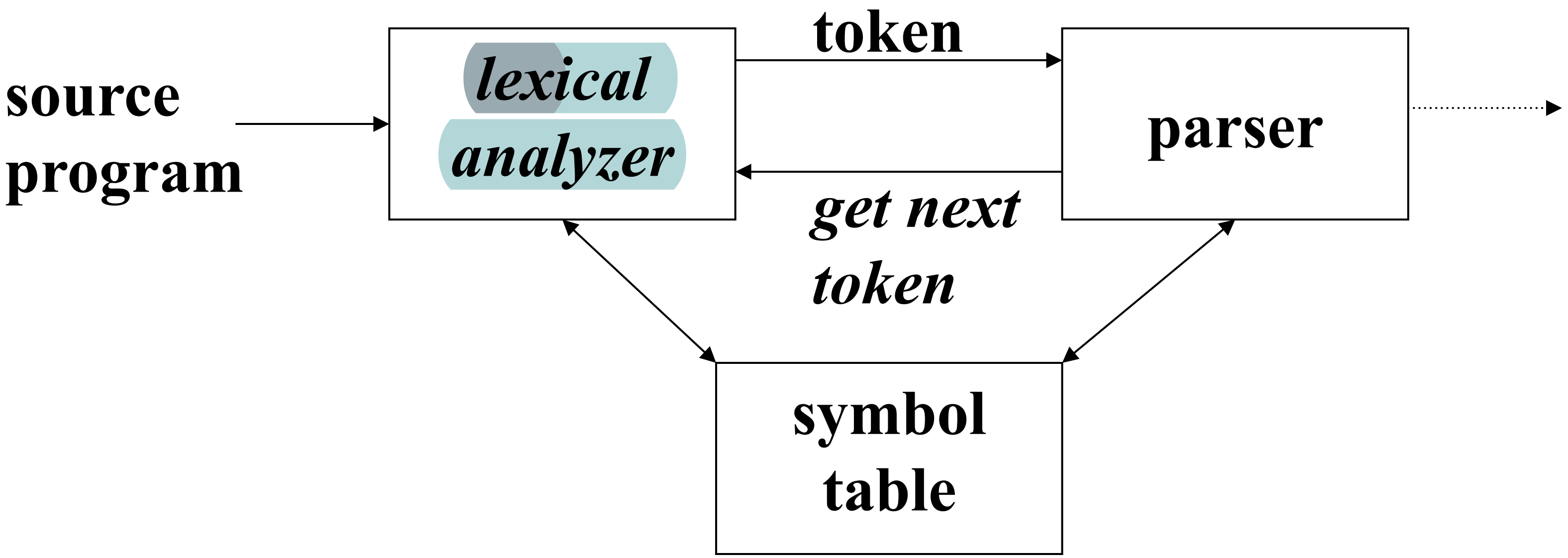
1. It read the characters and produces as output a sequence of tokens

2. It removes comments, white spaces (blank, tab, new line character) from the source program.

3. If lexical analyzer identifies any token of type identifier ,they are placed in symbol
table.

4. Lexical analyzer may keep track of the no. of new line characters seen, so that a line number can be associated with an error message.

Lexical Analyzer in Perspective


source program
lexical analyzer
token
parser
get next token
symbol table

Input Buffering

- The speed of lexical analyzer is the concern
- Lexical analysis needs to look ahead several characters before a match can be announced.
- Because of large amount of time consumption in moving characters, specialized buffering techniques have been developed to reduce the amount of overhead required to process an input character.
- Two buffer input scheme is useful when look ahead is necessary

- Buffer pairs

- Sentinels

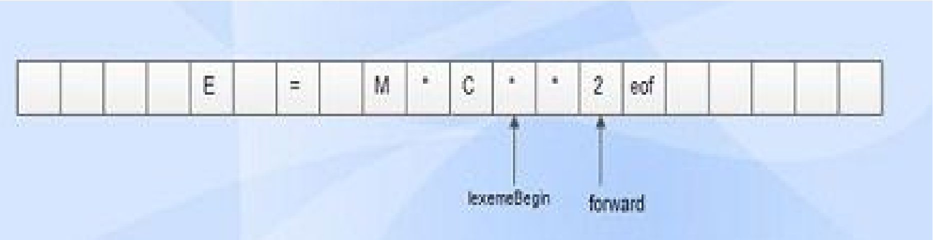


Fig. shows the buffer pairs which are used to hold the input data

Scheme

- Consists of two buffers, each consists of N-character size which are reloaded alternatively.
- N-Number of characters on one disk block, e.g., 4096.
- N characters are read from the input file to the buffer using one system read command.
- *eof* is inserted at the end if the number of characters is less than N.

Pointers

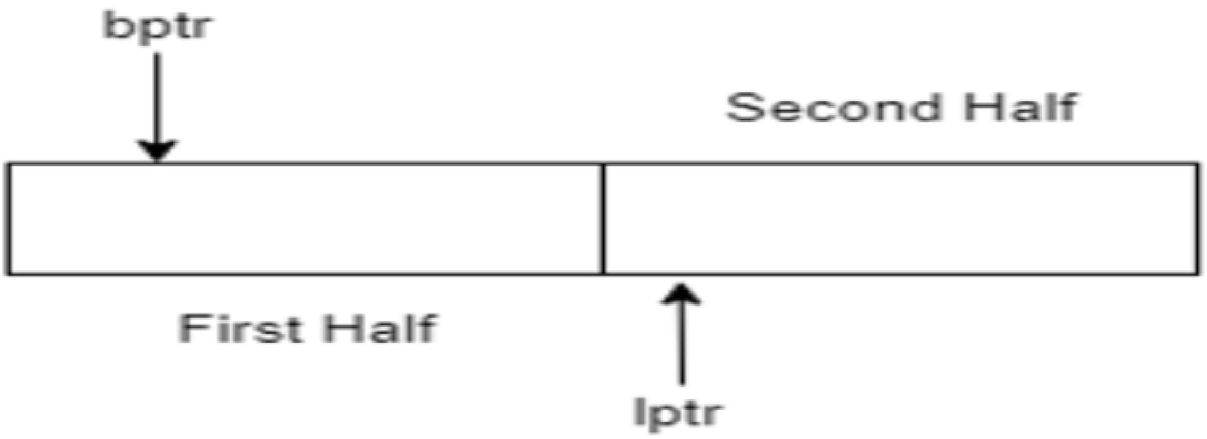
Two pointers *lexemeBegin* and *forward* are maintained.

lexeme Begin points to the beginning of the current lexeme which is yet to be found.

forward scans ahead until a match for a pattern is found.

- Once a lexeme is found, *lexemebegin* is set to the character immediately after the lexeme which is just found and *forward* is set to the character at its right end.
- Current lexeme is the set of characters between two pointers.

Two portions of a buffer can be separated. If you move the look Ahead cursor halfway through the first half, the second half will be filled with fresh characters to read. If you shift the look Ahead cursor to the right end of the second half's buffer, the first half will be filled with new characters, and so on.

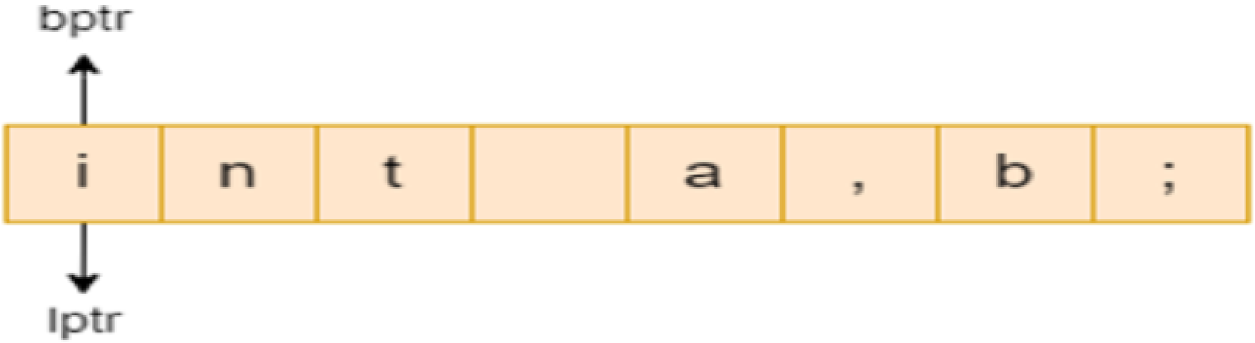


```
forward : = forward + 1;
if forward = eof then begin
        if forward at end of first half then begin
                reload second half;
                forward := forward + 1
        end
        else if forward at end of second half then begin
                reload first half
                move forward to beginning of first half
        end
        else /* eof within a buffer signifying end of input */
                terminate lexical analysis
end
```

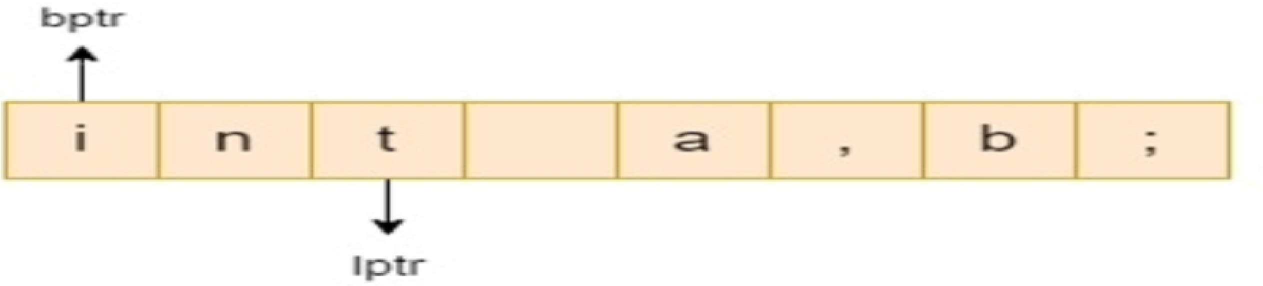
- Example: statement int a,b;

i	n	t		a	,	b	;

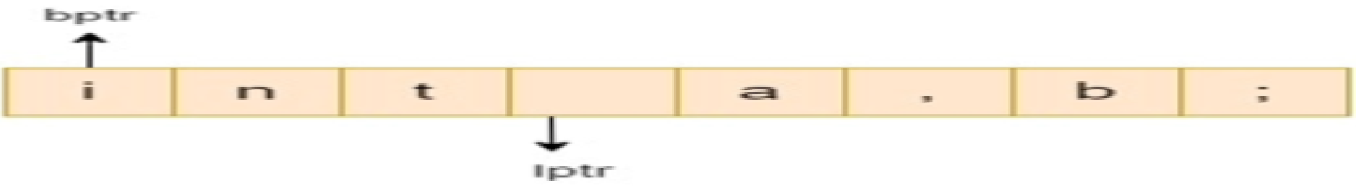
Both points begin at the start of the string that is saved in the buffer.



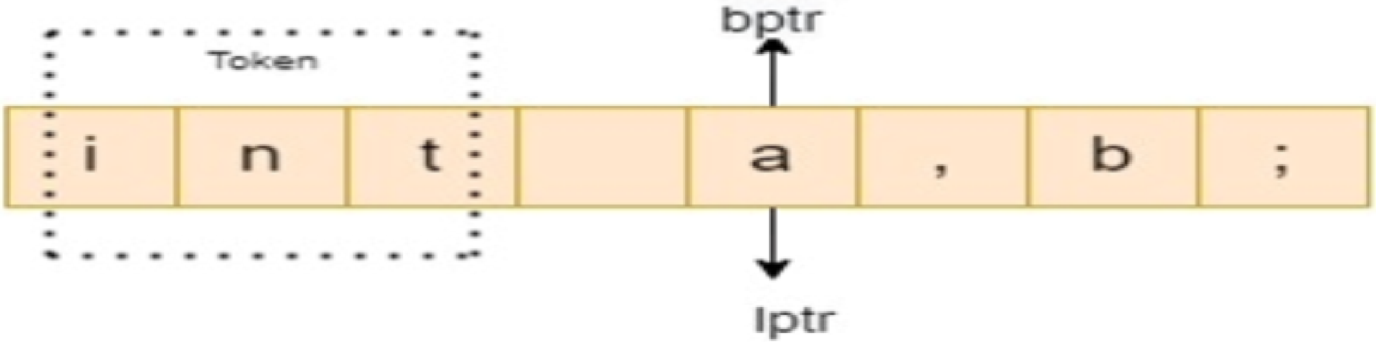
The Look Ahead Pointer examines the buffer until it finds the token.



Before the token ("int") can be identified, the character ("blank space") beyond the token ("int") must be checked.



Both pointers will be set to the next token ('a') after processing the token ("int"), and this procedure will be continued throughout the program.



Disadvantages of the scheme

- This scheme works well most of the time, but the amount of look ahead is limited.
- This limited lookahead may make it impossible to recognize tokens in situations where the distance that the forward pointer must travel is more than the length of the buffer.

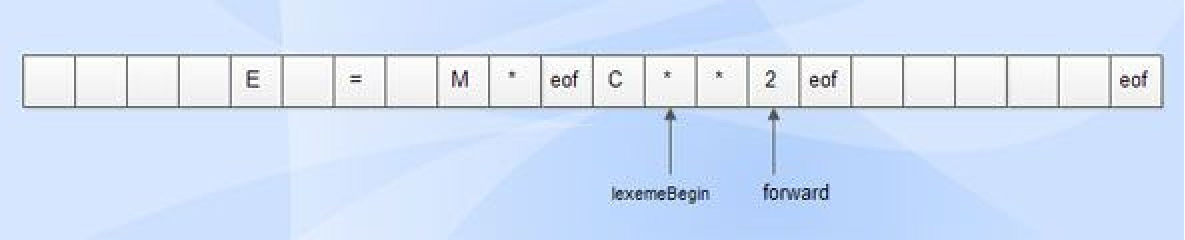
  (eg.) DECLARE (ARGl, ARG2, . . . , ARGn)
- It cannot determine whether the DECLARE is a keyword or an array name until the character that follows the right parenthesis.

Sentinels

- In the previous scheme, each time when the forward pointer is moved, a check is done to ensure that one half of the buffer has not moved off. If it is done, then the other half must be reloaded.
- Therefore the ends of the buffer halves require two tests for each advance of the forward pointer.

  Test 1: For end of buffer.

  Test 2: To determine what character is read.
- The usage of sentinel reduces the two tests to one by extending each buffer half to hold a sentinel character at the end.
- The sentinel is a special character that cannot be part of the source program. (*eof* character is used as sentinel).

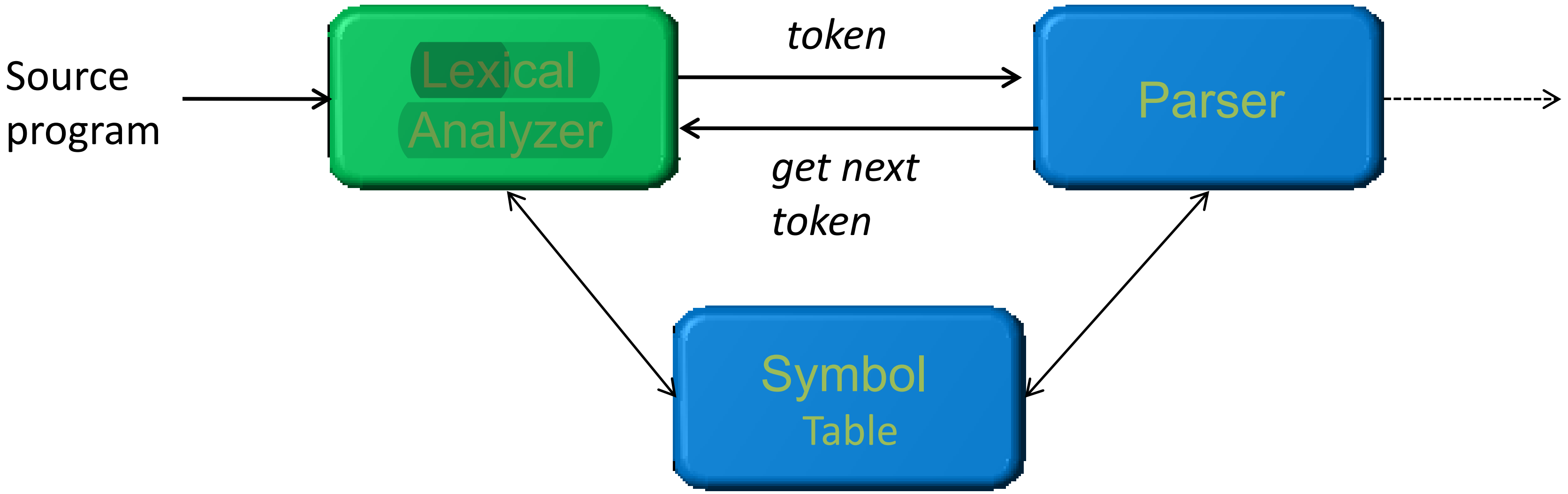


Advantages

- Most of the time, it performs only one test to see whether forward pointer points to an *eof*.
- Only when it reaches the end of the buffer half or *eof*, it performs more tests.
- Since N input characters are encountered between *eofs*, the average number of tests per input character is very close to 1.

Lexical Analyzer

- The Role of the Lexical Analyzer
  - Reads the input characters.
  - Produces as output a sequence of tokens to be used by the parser for syntax analysis.



Lexical Analyzer…

- Construction of a lexical analyzer requires the specification of tokens, patterns, and lexemes for the source language.
  - Tokens
    - Terminal symbols in the grammar for the source language
  - Patterns
    - Rules describing the set of lexemes that can represent a particular token in source programs.
  - Lexemes
    - Sequence of characters in the source program that are matched by the pattern for a token.

Lexical Analyzer

- Example:

Token	Pattern	Sample Lexemes
while	characters **w, h, i, l, e**	while
for	characters **f, o, r**	for
identifier	letter followed by letters and digits	total, result, a1, b4
integer	digit followed by digits	125, 23, 4567
string	characters surrounded by double quotes	“Hello World”, “a+b”

Lexical Analyzer…

- Attributes for Tokens
  - Required when more than one lexeme matches a pattern.
  - Lexical analyzer must provide additional information in such cases.
  - **Example:**
    - Consider the following C statement:
      - **X = Y - Z++**
    - The token names and associated attributes will be as shown below:
      - <**id**, pointer to symbol-table entry for X>
      - <**assign_op**>
      - <**id**, pointer to symbol-table entry for Y>
      - <**arith_op**, minus>
      - <**id**, pointer to symbol-table entry for Z>
      - <**incre_op**, post>

How to Construct a Lexical Analyzer?

- Construction of a Lexical Analyzer mainly involves two operations:
  - **Specification of Tokens**
    - This requires the knowledge of Languages and Regular Expressions.

- *Regular expressions* -A declarative way to express the pattern of any string over an alphabet or an algebraic way to describe languages.
- If E is a regular expression, then L(E) is the language it defines.
- A Language denoted by a regular expression is said to be *regular set.*

  - **Recognition of Tokens**
    - This requires the knowledge of Finite Automata.
    - Finite Automata is a recognizer that takes an input string & determines whether it’s a valid sentence of the language.

Specification of Tokens

- Alphabet
- Strings (words)
- Language
- Longest Match Rule
- Operations
- Notations
- Regular Expression

What is a Language?

- Alphabet (character class)
  - Denotes any finite set of symbols .
  - ∑ (sigma) is used to denote an alphabet
  - Example: {0,1} is binary alphabet
- String (sentence, or word)
  - Finite sequence of symbols drawn from an alphabet.
  - Example: { 00110, 01110101 }
- Language
  - Denotes any set of strings over some fixed alphabet.
  - Example: {{ 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111}

Operations on Languages

- Union, concatenation, closure, and exponentiation.
- Also called as regular operations.
- Definitions:

Operation	Definition and Notation
Union of *L* and *M*	**$L \cup M$ = { *s* | *s* is in *L* or *s* is in *M* }**
Concatenation of *L* and *M*	***L M* = { *st* | *s* is in *L* and *t* is in *M* }**
Kleene Closure of *L*	$L^* = \bigcup_{i=0}^{\infty} L^i$
Positive Closure of *L*	$L^+ = \bigcup_{i=1}^{\infty} L^i$

Operations on Languages…

- Example:
  - Let **L** be the set ***{A, B, . . . , Z, a, b, . . . , z}*** and
  - **D** the set ***{0, 1, . . . , 9}***, then
    - ***L U D*** *is the set of letters and digits.*
    - ***LD*** *is the set of strings consisting of a letter followed by a digit.*
    - ***L^4*** *is the set of all four-letter strings.*
    - ***L**** *is the set of all strings of letters, including ε, the empty string.*
    - ***L(L U D)**** *is the set of all strings of letters and digits beginning with a letter.*
    - ***D^+*** *is the set of all strings of one or more digits.*

Operations on Languages…

- Another Example:
  - Let the alphabet ∑ be the standard 26 letters (a,b,…,z}.
  - If ***A*** = {good, bad} and ***B*** = {boy, girl}, then what is the value of the following expressions?
    - **A U B**
    - **AB**
    - **A***
    - **A^+**

Longest Match Rule

- It states that the lexeme scanned should be determined based on the longest match among all available tokens.
- This is used to resolve ambiguities when deciding the next token in the input stream.
- During analysis of source code, the lexical analyzer will scan code letter by letter and when an operator, special symbol or whitespace is detected it decides whether a word is complete.

- Example: int intvalue;
- While scanning both lexemes till 'int', the lexical analyzer cannot determine whether it is a keyword *int* or the initials of identifier intvalue. The Longest Match Rule states that the lexeme scanned should be determined based on the longest match among all the tokens available.

- The lexical analyzer also follows rule priority where a reserved word, e.g., a keyword, of a language is given priority over user input. That is, if the lexical analyzer finds a lexeme that matches with any existing reserved word, it should generate an error.

What are Regular Expressions?

- Regular expressions are built using regular operations to describe languages.
- The value of a regular expression is a language.
- Examples:**$(0 \cup 1)\, 0^*$**
    - *The value of this expression is the language consisting of all strings starting with a 0 or a 1 followed by any number of 0s.*
    - *The value of this expression is the language consisting of all possible strings of 0s and 1s.*

What are Regular Expressions?

- Formal Definition

We say that **R** is a **regular expression** if **R** is	
1	**a** for some **a** in the alphabet Σ**,**
2	**ε,**
3	**ϕ**, empty language
4	($\mathbf{R_1}$ **U** $\mathbf{R_2}$), where $\mathbf{R_1}$ and $\mathbf{R_2}$ are regular expressions,
5	($\mathbf{R_1}\mathbf{R_2}$), where $\mathbf{R_1}$ and $\mathbf{R_2}$ are regular expressions, **or**
6	($\mathbf{R_1}$*****), where $\mathbf{R_1}$ regular expressions.

What are Regular Expressions?

- Examples:
  - Assume that the alphabet **B** is {0,1}.
  - What languages do the following regular expressions represent?
    - **0*10***
    - **B*1B***
    - **B*001B***
    - **$1^*(01^+)^*$**
    - **(BB)***
    - **(BBB)***
    - **$01 \cup 10$**
    - **$0B^*0 \cup 1B^*1 \cup 0 \cup 1$**

Regular Definitions

- If Σ is an alphabet of basic symbols, then a *regular definition* is a sequence of definitions of the form:

  $d_1 \to r_1$

  $d_2 \to r_2$

  .....

  $d_n \to r_n$

  where:

  - each d_i is a new symbol, not in Σ and not the same as any other of the d's, and
  - each r_i is a regular expression over the alphabet Σ U $\{d_1, d_2, . . . , d_{i-1})$.

Regular Definitions…

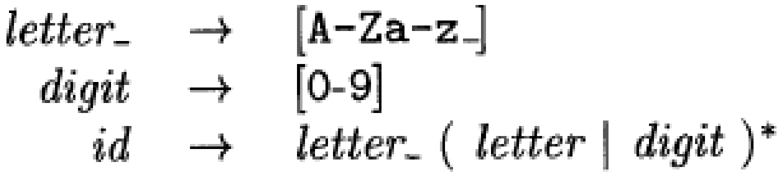
- Example:
  - C identifiers are strings of letters, digits, and underscores. The following is a regular definition for the language of C identifiers.
    - *letter_ → A | B | … | Z | a | b | … | z | _*
    - *digit → 0 | 1 | … | 9*
    - *id → letter_ (letter_ | digit)**

Extensions of Regular Expressions

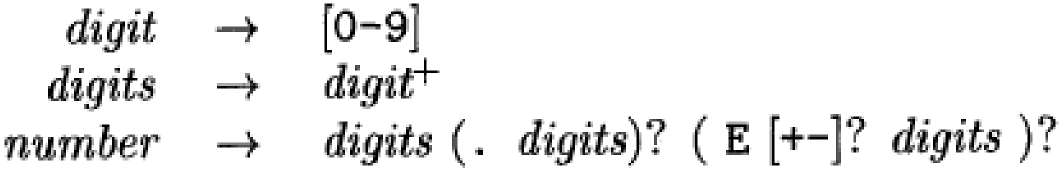
- One or more instances (+)
  - *letter+*
- Zero or one instance (?)
  - *letter (letter | digit)?*
- Character classes ([ ])
  - *[a-z]* instead of *a|b|c…|z*

Extensions of Regular Expressions…

- Other Examples:



$$\begin{array}{rcl} \mathit{letter\_} & \rightarrow & [\texttt{A-Za-z\_}] \\ \mathit{digit} & \rightarrow & [0\text{-}9] \\ \mathit{id} & \rightarrow & \mathit{letter\_} \; (\; \mathit{letter} \mid \mathit{digit} \;)^* \end{array}$$



$$\begin{array}{rcl} \mathit{digit} & \rightarrow & [0\text{-}9] \\ \mathit{digits} & \rightarrow & \mathit{digit}^+ \\ \mathit{number} & \rightarrow & \mathit{digits} \; (.\; \mathit{digits})? \; (\; \texttt{E} \; [+-]? \; \mathit{digits} \;)? \end{array}$$

Review of Finite Automata (FA)

Non-Deterministic: Has more than one alternative action for the same input symbol Non-Deterministic Finite Automata (NFAs) easily represent regular expression, but are somewhat less precise.

Deterministic : Has at most one action for a given input symbol. Deterministic Finite Automata (DFAs) require more complexity to represent regular expressions, but offer more precision.

Both types are used to recognize regular expressions.

Construction of a Lexical Analyzer: Recognition of Tokens

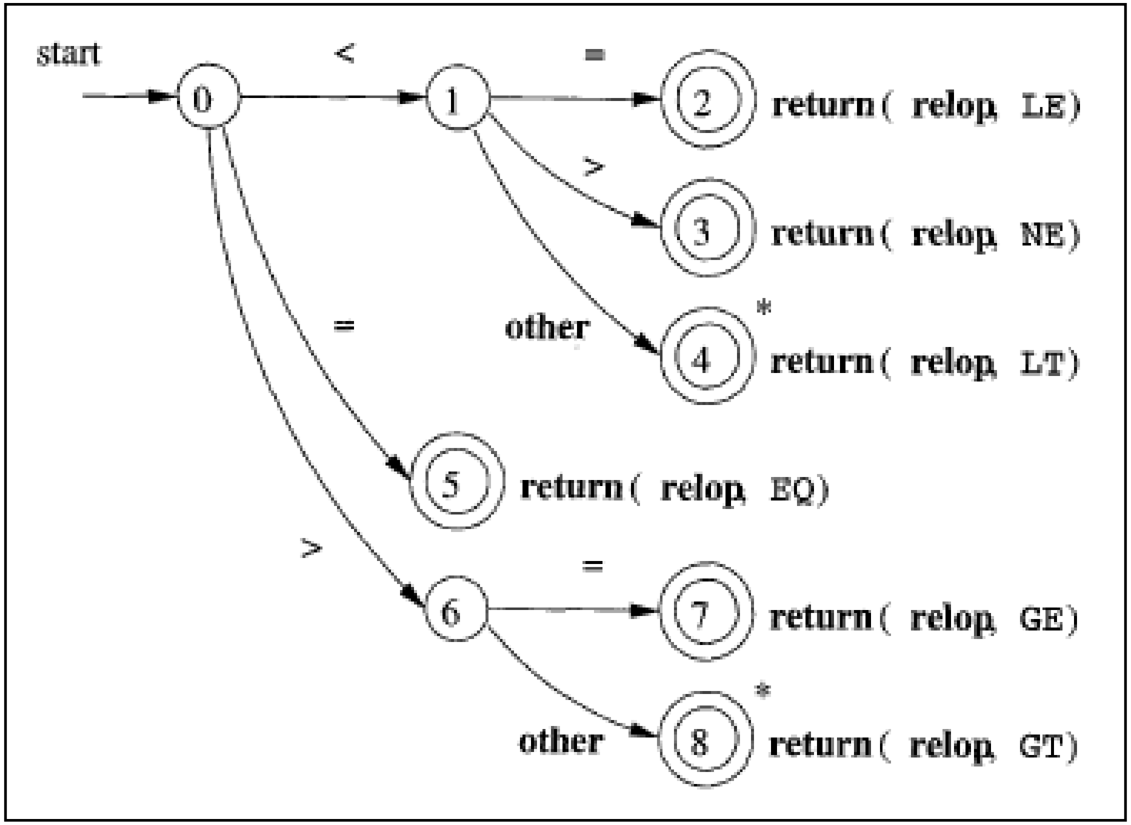
- Convert regular expressions into transition diagrams.
- Convert transition diagrams into a lexical analyzer program.

What are Transition Diagrams?

- These are similar to a DFA.
- These contain
  - A start state,
  - Zero or more intermediate states,
  - One or more accepting states or final states, and
  - Edges labeled with one or more symbols.
- Converting regular expressions into a transition diagram is an intermediate step in the construction of a lexical analyzer.

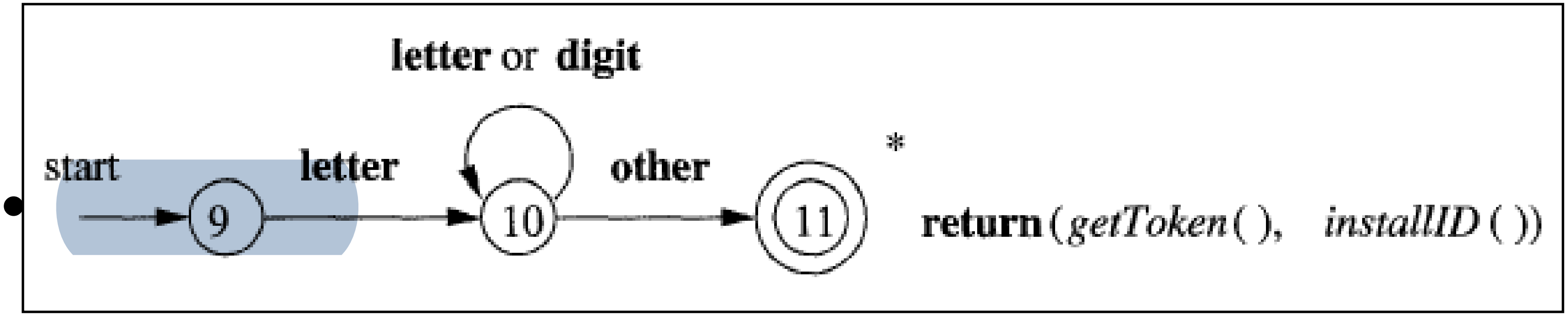
Converting Regular Expressions into Transition Diagrams

- Example 1:
  - Transition diagram for **relational operators**

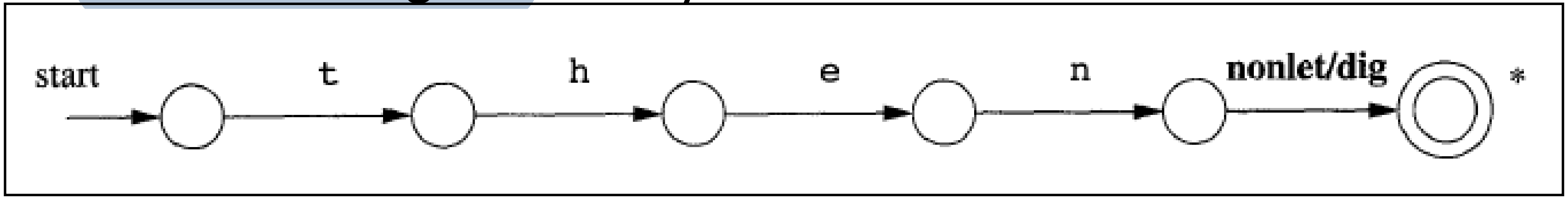


Converting Regular Expressions into Transition Diagrams…

- Example 2:
  - Transition diagram for **reserved words** and **identifiers**

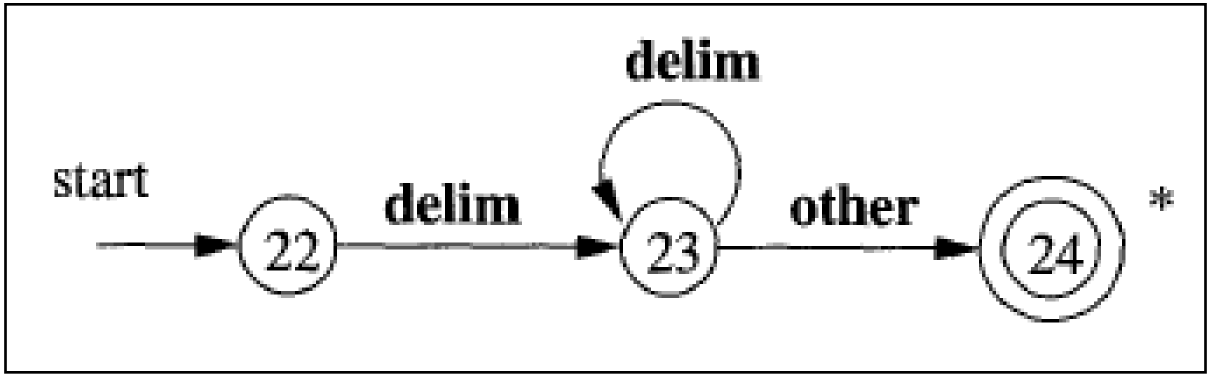


  - Transition diagram for keyword ***then***

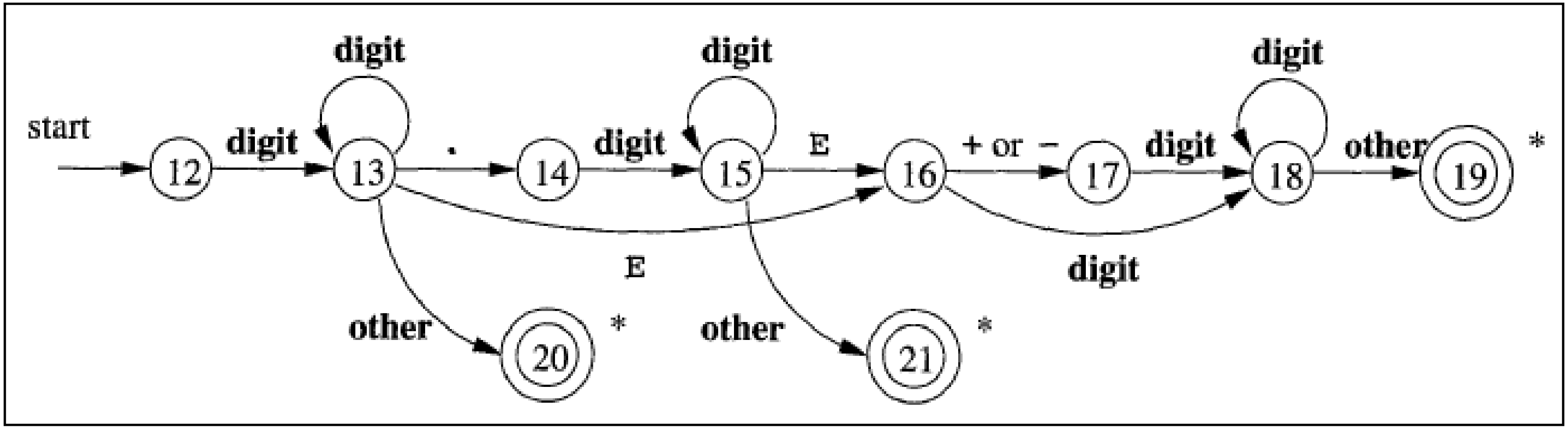


Converting Regular Expressions into Transition Diagrams

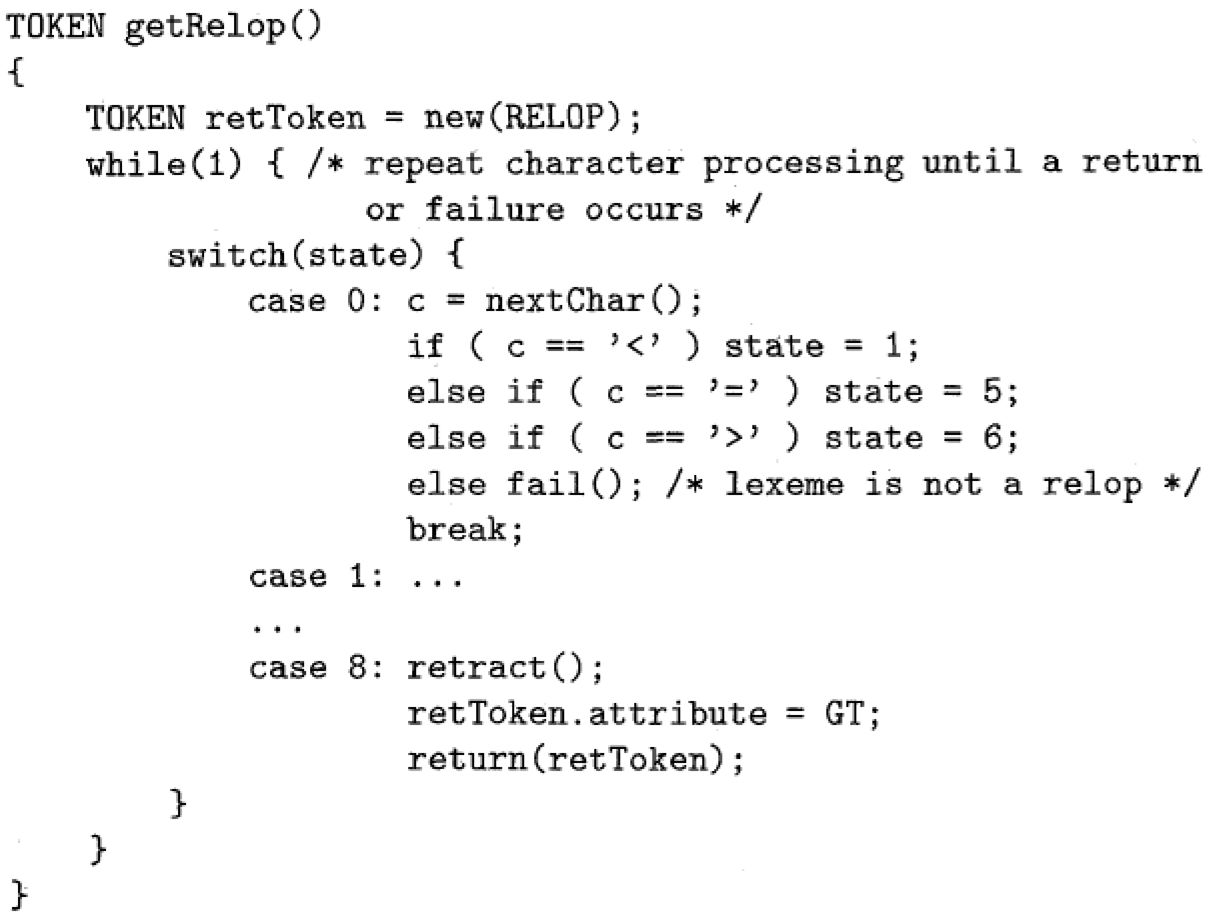
- Example 4:
  - Transition diagram for **white space**



- Example 5:
  - Transition diagram for **unsigned numbers**

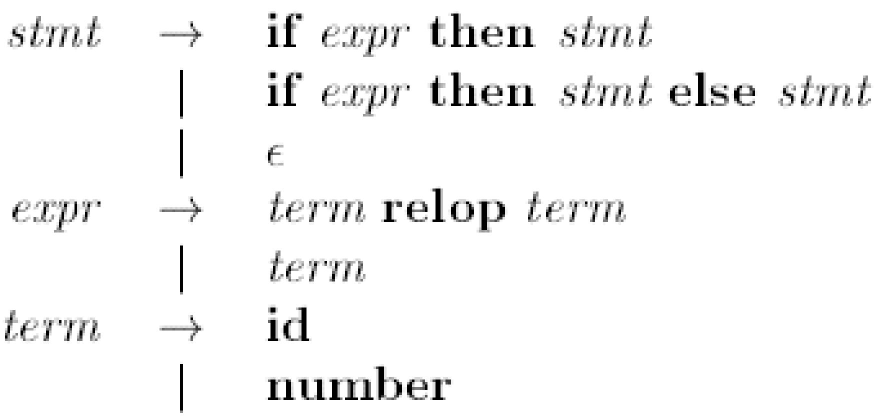


Converting Transition Diagram into a Program

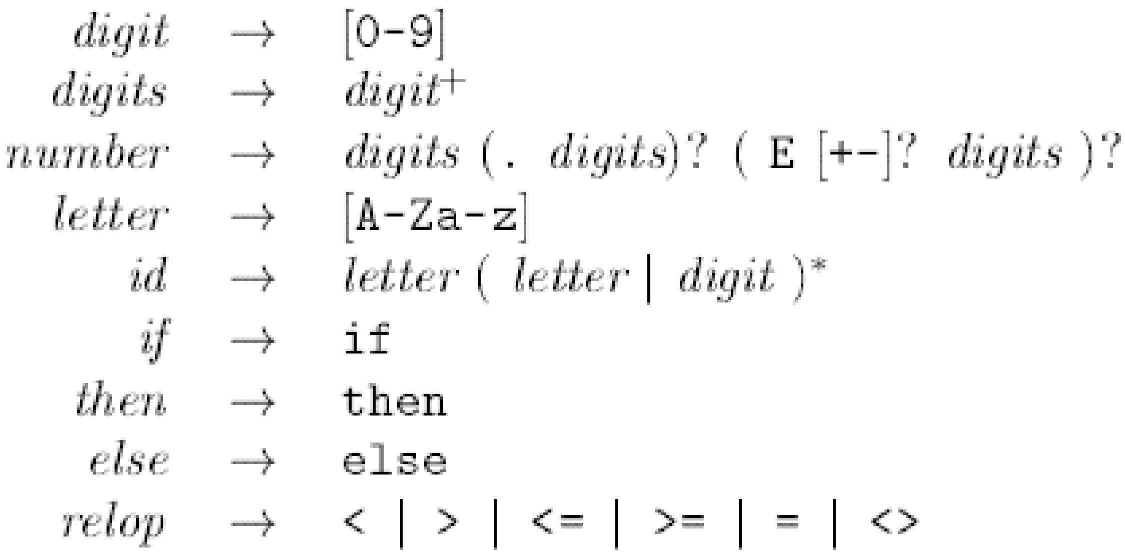


```
TOKEN getRelop()
{
    TOKEN retToken = new(RELOP);
    while(1) { /* repeat character processing until a return
                  or failure occurs */
        switch(state) {
            case 0: c = nextChar();
                    if ( c == ’<’ ) state = 1;
                    else if ( c == ’=’ ) state = 5;
                    else if ( c == ’>’ ) state = 6;
                    else fail(); /* lexeme is not a relop */
                    break;
            case 1: ...
            ...
            case 8: retract();
                    retToken.attribute = GT;
                    return(retToken);
        }
    }
}
```

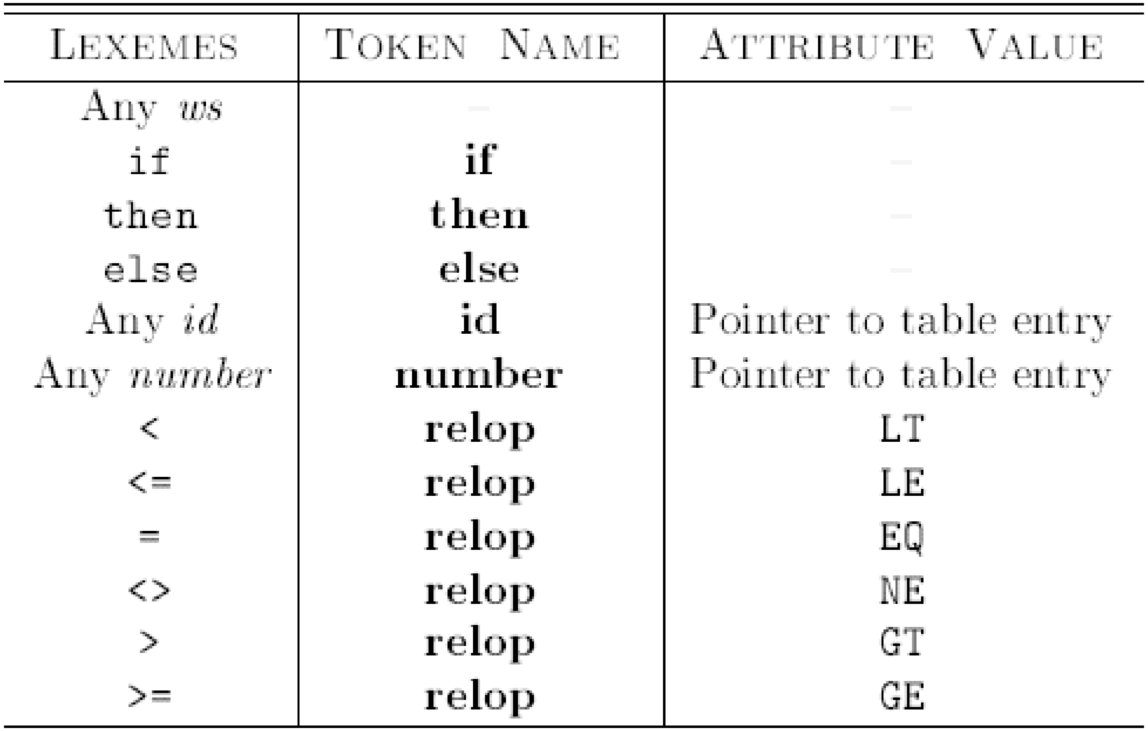
Recognition of Tokens



$$\begin{array}{lcl}
\mathit{stmt} & \to & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt} \\
 & | & \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt}\ \textbf{else}\ \mathit{stmt} \\
 & | & \epsilon \\
\mathit{expr} & \to & \mathit{term}\ \textbf{relop}\ \mathit{term} \\
 & | & \mathit{term} \\
\mathit{term} & \to & \textbf{id} \\
 & | & \textbf{number}
\end{array}$$



$$\begin{array}{rcl}
\mathit{digit} & \to & \texttt{[0-9]} \\
\mathit{digits} & \to & \mathit{digit}^{+} \\
\mathit{number} & \to & \mathit{digits}\ (\texttt{.}\ \mathit{digits})?\ (\ \texttt{E}\ \texttt{[+-]}?\ \mathit{digits}\ )? \\
\mathit{letter} & \to & \texttt{[A-Za-z]} \\
\mathit{id} & \to & \mathit{letter}\ (\ \mathit{letter}\ |\ \mathit{digit}\ )^{*} \\
\mathit{if} & \to & \texttt{if} \\
\mathit{then} & \to & \texttt{then} \\
\mathit{else} & \to & \texttt{else} \\
\mathit{relop} & \to & \texttt{<}\ |\ \texttt{>}\ |\ \texttt{<=}\ |\ \texttt{>=}\ |\ \texttt{=}\ |\ \texttt{<>}
\end{array}$$

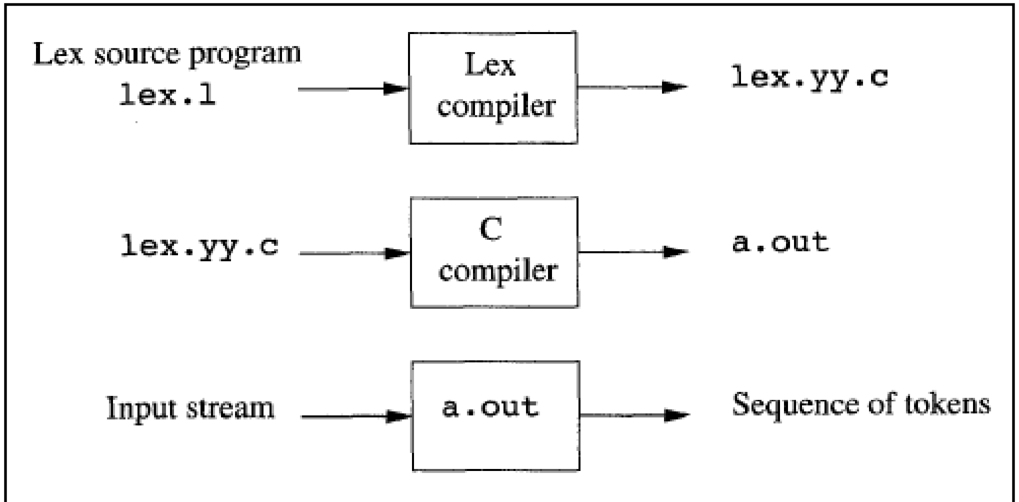


LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
`<`	**relop**	`LT`
`<=`	**relop**	`LE`
`=`	**relop**	`EQ`
`<>`	**relop**	`NE`
`>`	**relop**	`GT`
`>=`	**relop**	`GE`

Lexems, their token and attribute values

Lex Tool

- Lex is a Lexical Analyzer Generator.
- Used to create a lexical analyzer.
- Lex Language is the input notation for Lex tool.
- Lex Compiler is used to compile Lex programs.



Creating a Lexical Analyzer with Lex

Structure of Lex Programs

- A Lex program has the following form:

  declarations

  %%

  translation rules

  %%

  auxiliary functions

Structure of Lex Programs…

- Declaration Section
  - Includes declarations of
    - Variables
    - Manifest constants, and
      - identifiers declared to represent a constant
    - Regular definitions

Structure of Lex Programs – contd.

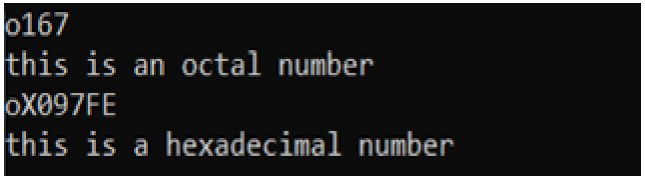
- **Translation Rules Section**
  - Each translation rule has the following form:
    - *Pattern { Action }*
  - Each pattern is a regular expression
    - May use regular definitions of declaration section
  - Actions are fragments of code typically written in C

- **Auxiliary Functions Section**
  - Holds additional functions used in actions

Example- Program to identify octal or hexadecimal number using LEX

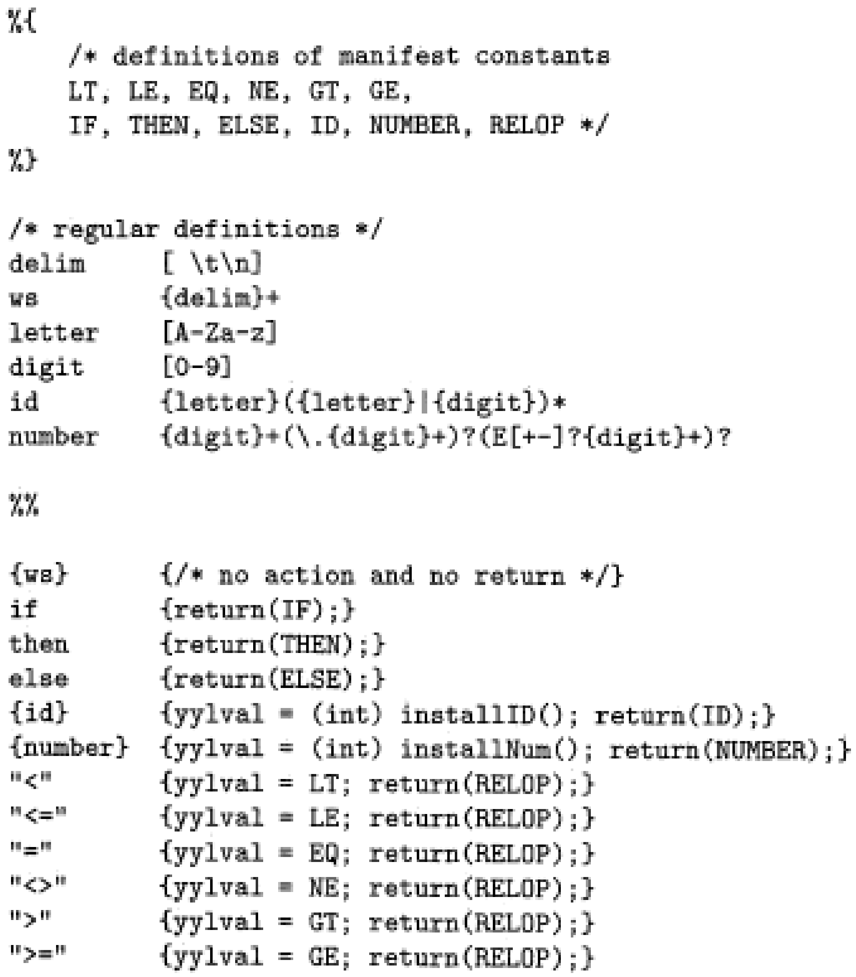
The octal number system is a base-8 number system and uses the digits 0 - 7 to represent numbers. The hexadecimal number system is a base-**16** number system and uses the digits 0 - 9 along with the letters A - F to represent numbers.

```
%{
    /*program to identify octal and hexadecimal numbers*/
%}
Oct [o][0-7]+
Hex [o][x|X][0-9A-F]+
%%
{Hex} printf("this is a hexadecimal number");
{Oct} printf("this is an octal number");
%%
main()
{
printf("Enter the input\n");
yylex();             /*returns a value indicating the type of token that has been obtained*/
}
int yywrap()      /*is called by lex when input is exhausted, return 1 if you are done or 0 if more processing is required*/
{
return 1;
}
```



```
o167
this is an octal number
oX097FE
this is a hexadecimal number
```

Example Lex Program

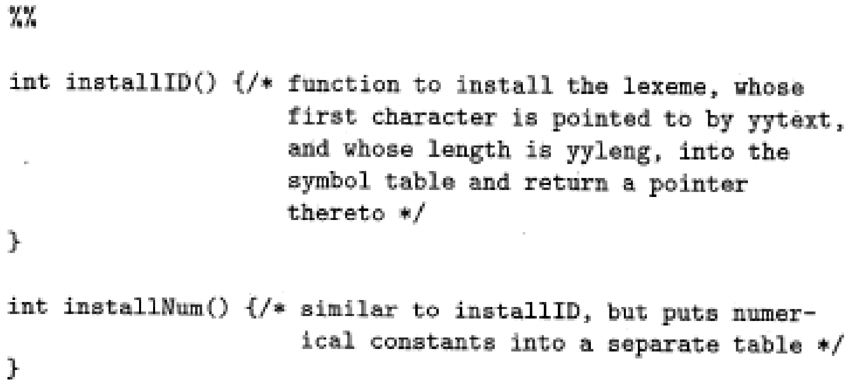


```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim     [ \t\n]
ws        {delim}+
letter    [A-Za-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
number    {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%

{ws}      {/* no action and no return */}
if        {return(IF);}
then      {return(THEN);}
else      {return(ELSE);}
{id}      {yylval = (int) installID(); return(ID);}
{number}  {yylval = (int) installNum(); return(NUMBER);}
"<"       {yylval = LT; return(RELOP);}
"<="      {yylval = LE; return(RELOP);}
"="       {yylval = EQ; return(RELOP);}
"<>"      {yylval = NE; return(RELOP);}
">"       {yylval = GT; return(RELOP);}
">="      {yylval = GE; return(RELOP);}
```

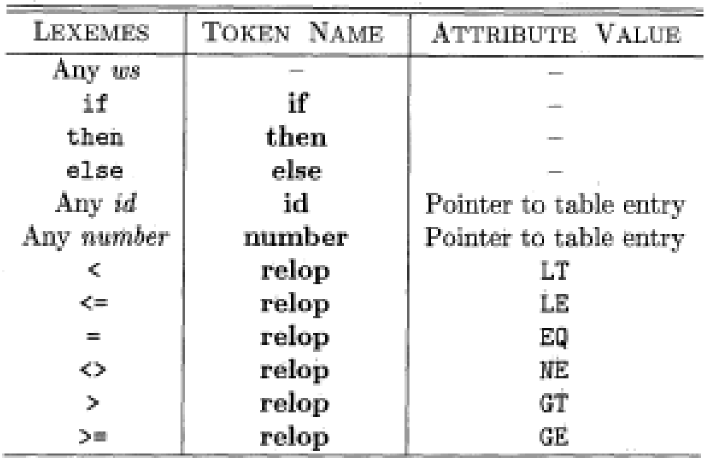


```
%%

int installID() {/* function to install the lexeme, whose
                    first character is pointed to by yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer
                    thereto */
}

int installNum() {/* similar to installID, but puts numer-
                     ical constants into a separate table */
}
```

Lex Program for the following Tokens



LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
if	if	–
then	then	–
else	else	–
Any *id*	id	Pointer to table entry
Any *number*	number	Pointer to table entry
<	relop	LT
<=	relop	LE
=	relop	EQ
<>	relop	NE
>	relop	GT
>=	relop	GE