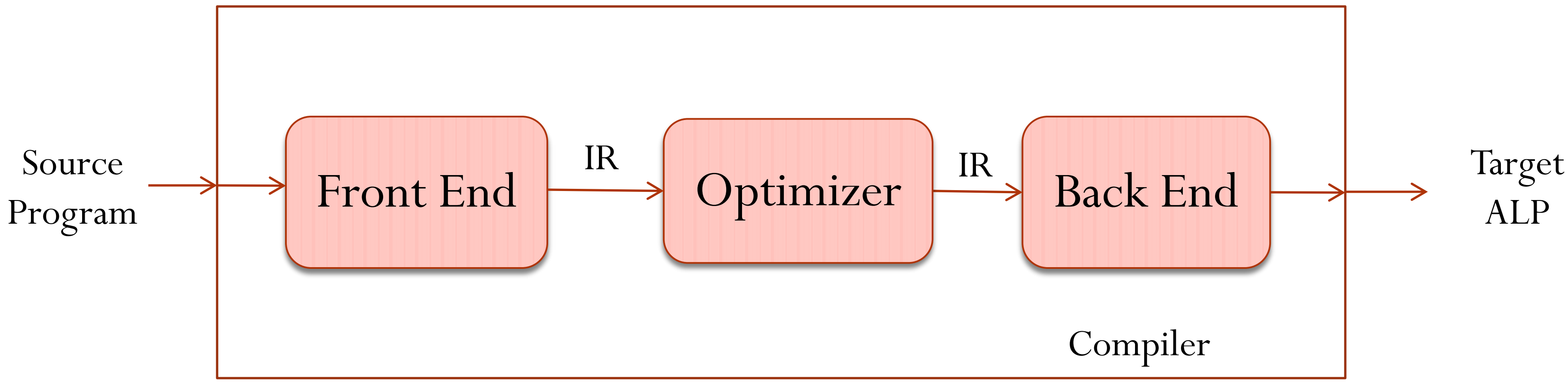
Compilers

Topic: Lexical Analysis

Monsoon 2011, IIIT-H, Suresh Purini

ACK: Some slides are based on Keith Cooper’s CS412 at Rice University

The Front End

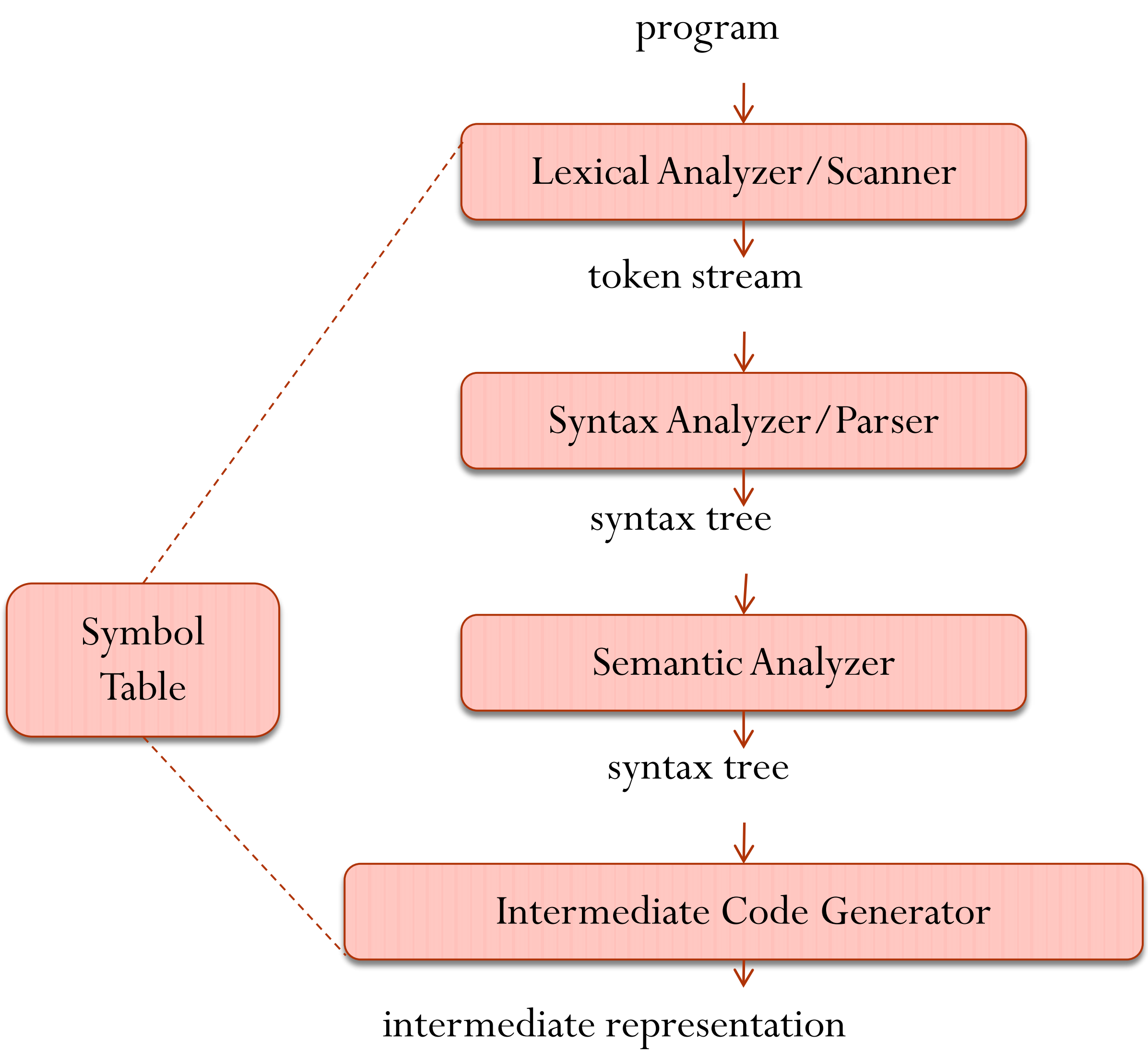


The purpose of the front end is to deal with the input language

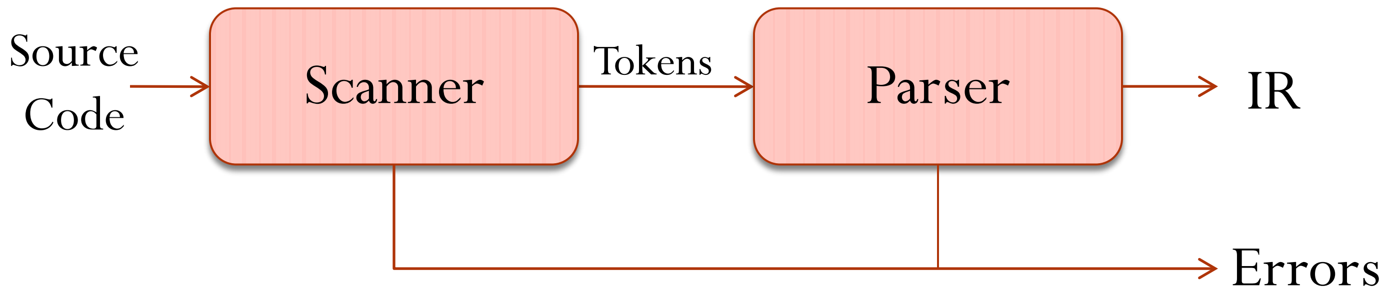
- Perform a membership test: code ∈ source language?
- Is the program well-formed (semantically) ?
- Build an IR version of the code for the rest of the compiler

The front end is not monolithic

The Front End



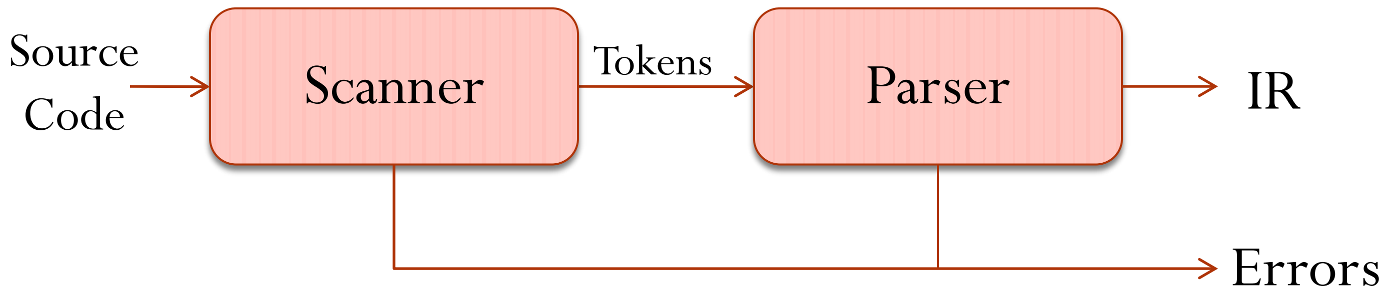
The Front End: Scanner and Parser



Scanner

- Maps character stream into words—the basic unit of syntax
- Produces pairs — a word & its part of speech
  - x = x + y ; becomes <id,x> = <id,x> + <id,y> ;
  - word ≅ lexeme, part of speech ≅ token type, pair ≅ a token
- Typical tokens include number, identifier, +, –, new, while, if

The Front End: Scanner and Parser



Parser

- Takes as input a stream of tokens
- Checks if the stream of tokens constitute a syntactically valid program of the language
- If the input program is syntactically correct
  - Output a concrete representation of program (like AST)
- If the input program has syntactic errors
  - Outputs relevant diagnostic information

Syntax Specification Using Context Free Grammars

1. Goal $\longrightarrow$ Expr
2. Expr $\longrightarrow$ Expr Op Term | Term
3. Term $\longrightarrow$ number | id
4. Op $\longrightarrow$ + | -

S = Goal (Start Symbol)

T = { number, id, +, - }

N = { Goal, Expr, Term, Op }

Question: Given a stream of tokens (read terminals) and the syntax specification in the form a CFG, how can the parser check the syntactic correctness of the source code?

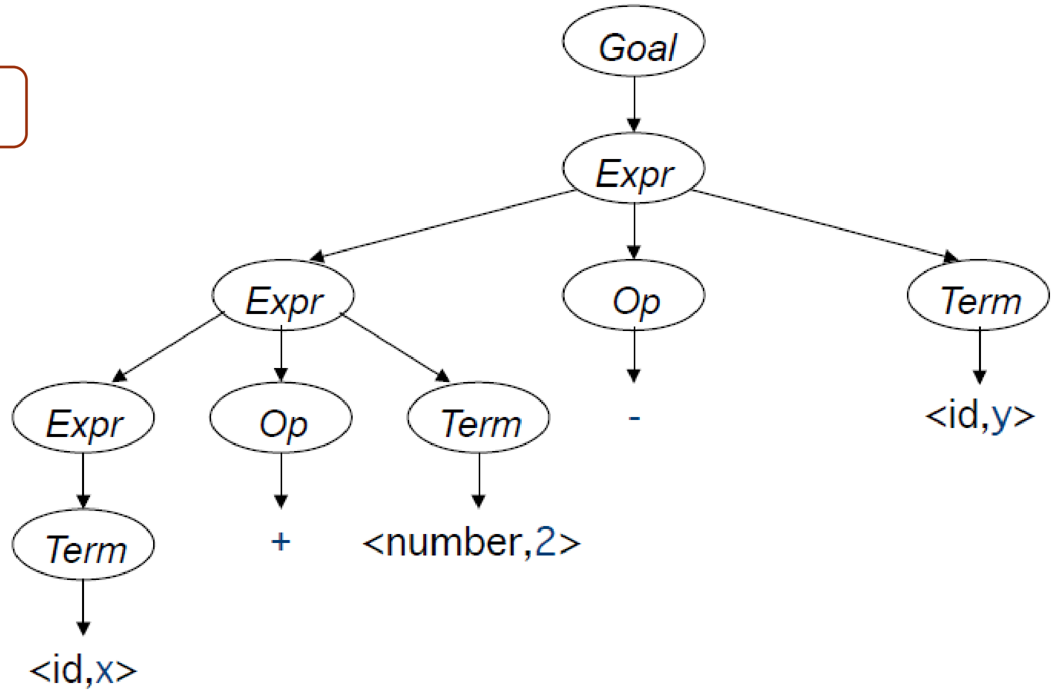
Formally, a grammar G = (S,N,T,P)

- S is the start symbol
- N is a set of non-terminal symbols
- T is a set of terminal symbols or words
- P is a set of productions or rewrite rules

Parse Trees

- Any valid sentence of the language can be represented by a corresponding Parse Tree or Syntax Tree

x + 2 - y

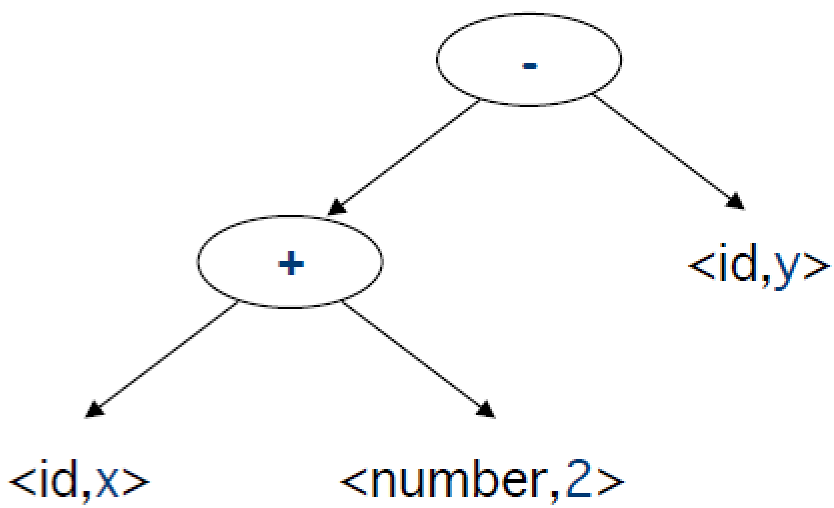


- A Parse Tree contains lot of derivation information not needed by the compiler passes down the lane.

Abstract Syntax Trees (ASTs)

- Compilers often use an abstract syntax tree instead of a parse tree
- The AST summarizes grammatical structure, without including detail about the derivation

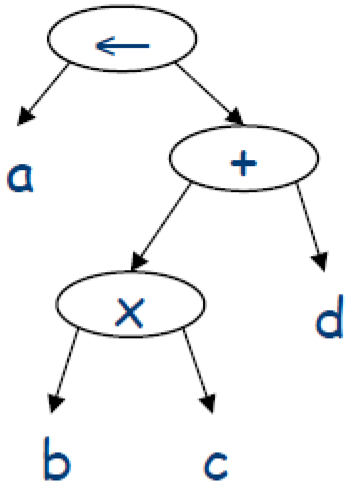
x + 2 - y



- This is much more concise
- ASTs are one kind of intermediate representation (IR)

Abstract Syntax Trees

a ← b x c + d



Lexical Tokens

1. Goal $\longrightarrow$ Expr
2. Expr $\longrightarrow$ Expr Op Term | Term
3. Term $\longrightarrow$ Number | Id
4. Op $\longrightarrow$ + | -
5. Id $\longrightarrow$ Alpha Id | Alpha
6. Alpha $\longrightarrow$ a | b | c | …. | z
7. Number $\longrightarrow$ Number Digit | Digit
8. Digit $\longrightarrow$ 0 | 1 | …. | 9

1. Goal $\longrightarrow$ Expr
2. Expr $\longrightarrow$ Expr op Term | Term
3. Term $\longrightarrow$ number | id

   (+, - are possible ops)

1. Goal $\longrightarrow$ Expr
2. Expr $\longrightarrow$ Expr + Term |
   | Expr – Term | Term
3. Term $\longrightarrow$ number | id

- A Token is a fundamental (atomic) Syntactic unit in a language.
- **It depends on how the grammar for the language is defined though.**

Micro-Syntax and Macro-Syntax

- Micro-Syntax – The rules that govern the lexical structure of a language is called the micro-syntax of the language.
  - Specified using Regular Expressions
  - Implemented using Finite State Machines
- Macro-Syntax – The grammar itself is called the macro-syntax of the language.
  - Specified using CFGs
  - Implemented using Push Down Automata

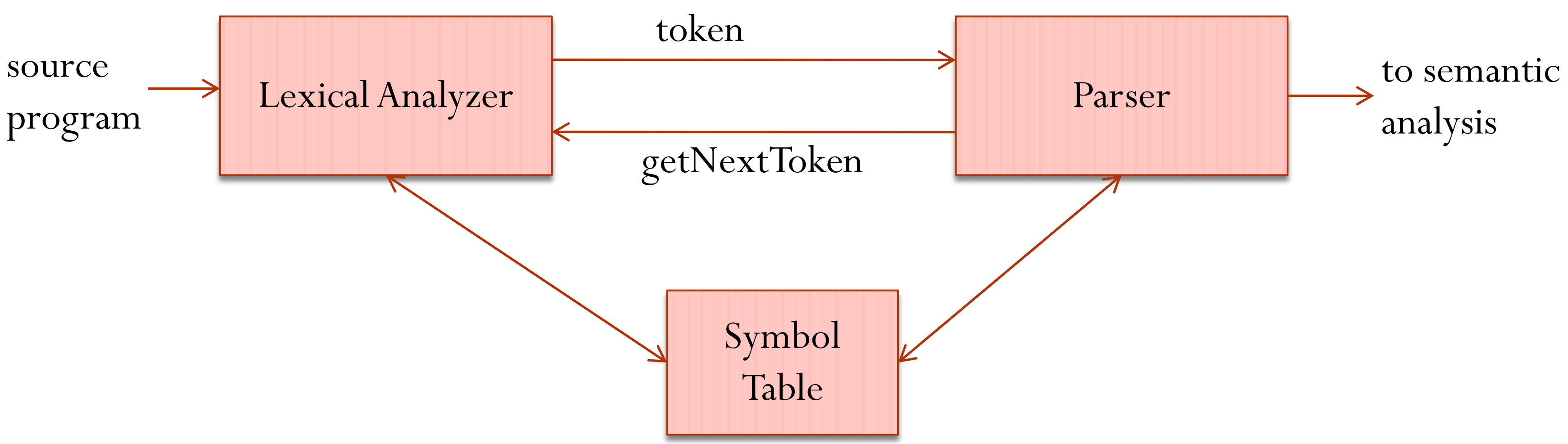
Micro-syntax versus Macro-syntax

Question: Why can't we encode the micro-syntax of the language into the grammar for the macro syntax?

Don't lift a needle with a crane?

- Separation of micro-syntax gives a clean grammar for the core programming language constructs.
  - Grammar need not keep track of nitty-gritty details which are taken care of cleanly and efficiently in lexical analyzer.
- Every grammar rule dropped shrinks the size of parser.
  - Parsing is more harder and slower than lexical analysis.

Lexical Analyzer and Parser

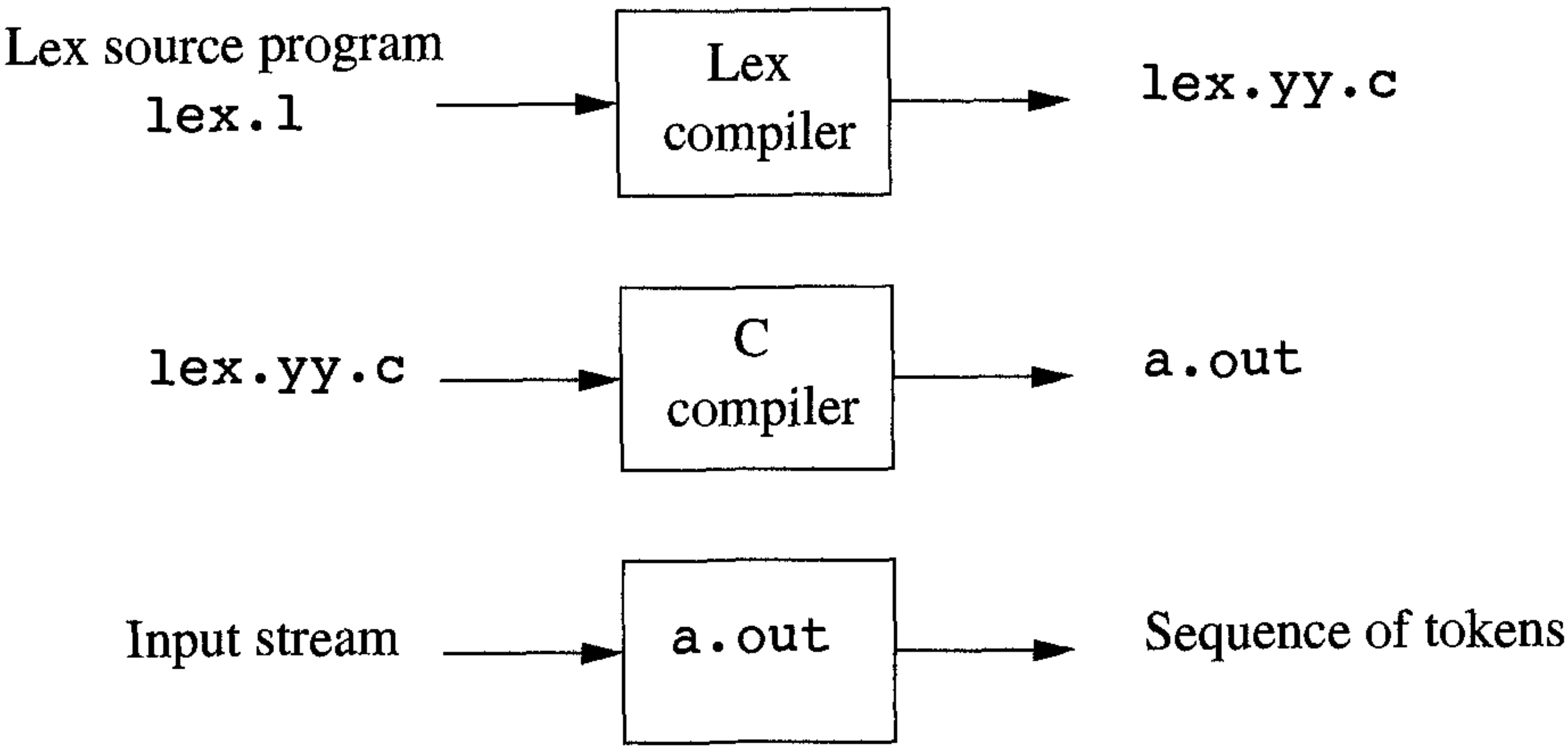


Logical Design of a Lexical Analyzer

For a given programming language

1. First Step: Identify all the possible Tokens that can occur in any valid program.
   - We need to consult the grammar for the programming language to identify the Tokens.
2. Second step: Specify the tokens using regular expressions.
3. Third Step:
   a) Either hand-write the lexical analyzer or
   b) Use lexical analyzer generator tool like flex or jflex.

Lexical Analyzer Generator Lex



Structure of a Lex Program

declarations

%%

translation rules → Pattern { Action }

%%

auxiliary functions

Lex – Sample Translation Section

```
/* 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);}
```

Two Simple Rules

1. Return the longest matching lexeme (token).
2. Token types are prioritized according to their occurrence in the lex file.

Pattern Matching Rules

- Always prefer a longer prefix to a shorter prefix
- If the longest possible prefix matches two or more patterns, prefer the pattern listed first in the Flex program

Handwritten or Auto-Generated Scanners

- In practice, many people still write scanners by hand. Even if you intend to hand-code a scanner, writing down the specification and understanding the automata for it is extremely useful.
- Ex: llvm (clang)
- Our Next Goal: Learn the Theory and Practice behind Automatic Scanner Generators.

Lexical Analyzer Design

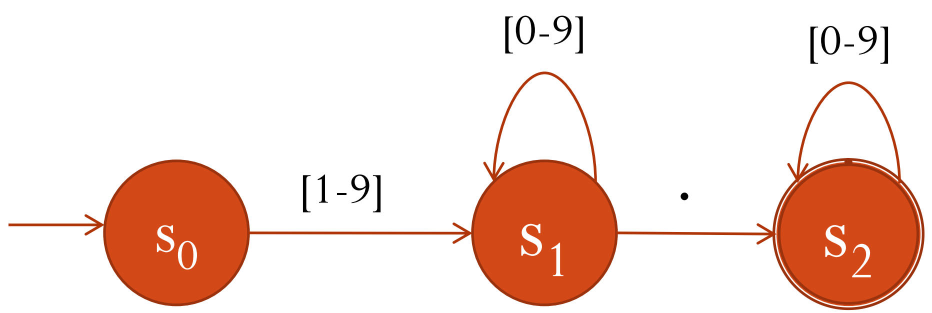
First Problem: Given a RE R, build a C function which checks whether an input string is described by the RE R.

1. Convert the RE into an NFA.
2. Convert NFA to DFA.
3. Minimize the DFA.
4. Translate the DFA into an equivalent program function.

Time Complexity of the Resulting Function: O(t) where t is the input string length.

Encoding a DFA into a Program Function

Example: FLOAT_LITERAL $[1\text{-}9][0\text{-}9]^*.[0\text{-}9]^*$

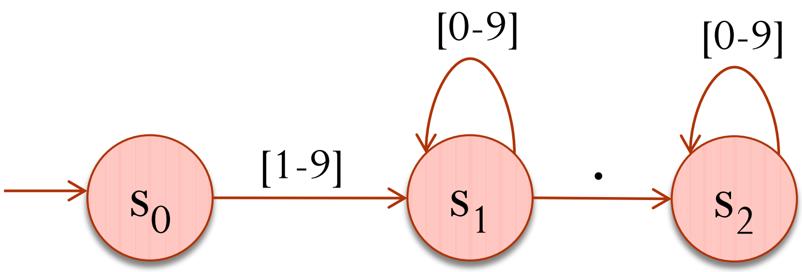


δ	.	0	1	2	3	4	5	6	7	8	9
s_0	s_e	s_e	s_1	s_1	s_1	s_1	s_1	s_1	s_1	s_1	s_1
s_1	s_2	s_1	s_1	s_1	s_1	s_1	s_1	s_1	s_1	s_1	s_1
s_2	s_e	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2
s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e

Note: We can compress the table by identifying common columns.

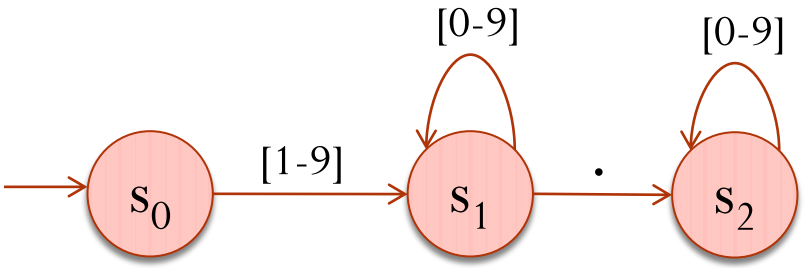
Table Driven Approach for Encoding DFAs

```
IsFloatLiteral(String str)
Begin
  state = s0
  for i = 1 to str.length
  do
     state = nextState(state, str[i])
      if ( state = se ) then return reject
  end for
  if ( state ∈ F )
     then return accept
     else  return reject
End
```



1) Only the Transition Function Table used by the nextState function changes to recognize another token.
2) Well the set of Final States F also changes.

Direct Coding Approach




```
IsFloatLiteral(String str)
Begin
  i = 1
  s0: if ( i > str.length )  then return reject
      if ( '1' ≤ str[i] ≤ '9' ) then goto s1
       return reject
 s1:  i = i + 1
     if ( i > str.length ) then goto return reject
     if ( '0' ≤ str[i] ≤ '9' ) then goto s1
     if ( str[i] = '.' ) then goto s2
       else return reject
 s2:  i = i + 1
     if ( i > str.length ) then goto return accept
     if ( '0' ≤ str[i] ≤ '9' ) then goto s2
       else  return reject
End
```

Table-Driven versus Direct-Coded Approach

Table Driven Approach	Direct Coded Approach
One multiplication operation per terminal symbol.	No Multiplication operation.
One branch instruction per terminal. But these branches are highly predictable (loop back).	One branch instruction per terminal. But predicting the branch targets could be hard.
Good code locality.	Code locality?
Data Locality?	Data Locality?
…….	…….

Lexical Analyzer Design

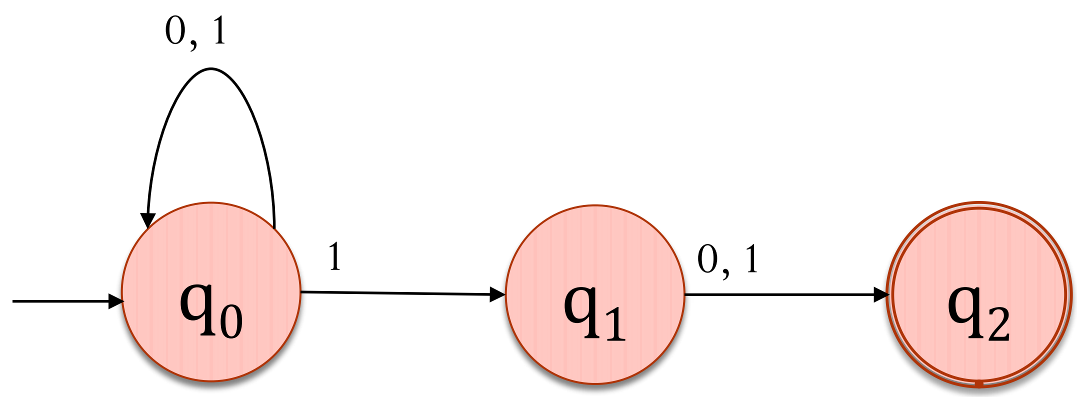
Second Problem: Given an sequence of REs R_1, …., R_k , construct a function which takes a string as input and outputs the first RE R_i describing the input string.

Time Complexity: O(kt)

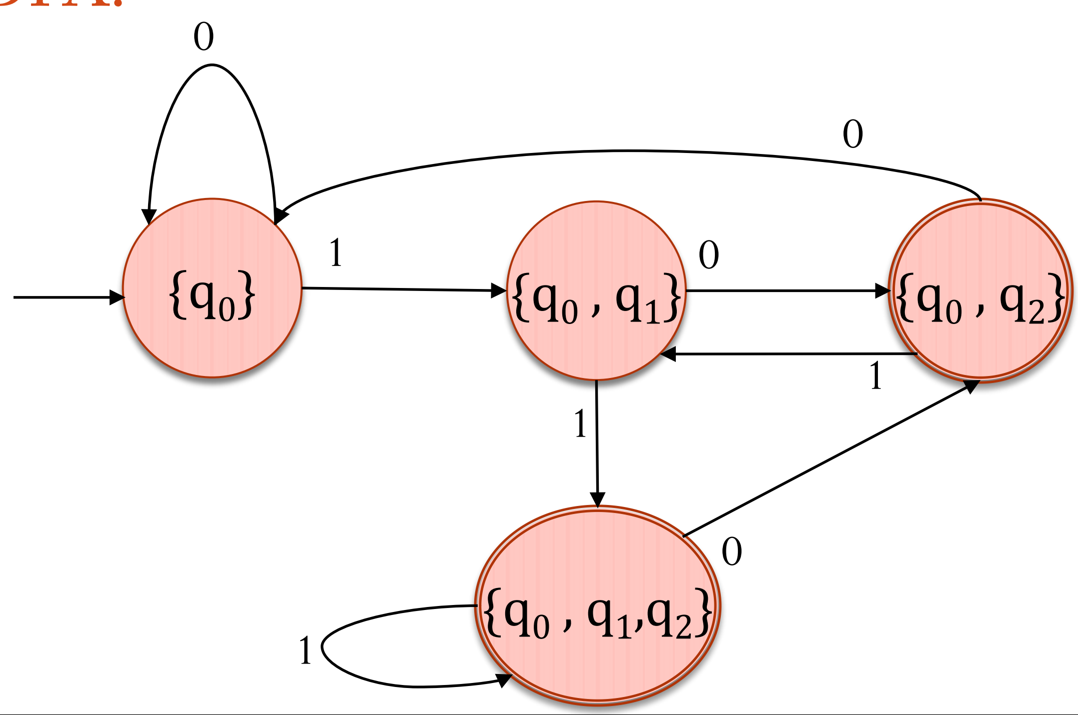
Q: Can we design a O(t) algorithm for the checkString function?

```
checkString(char *str)
{
 if ( IsR1(str) = True) return 1;
  ………………………..
 if ( IsRk(str) = True) return k;
 return -1;
}
```

Recall: NFA to DFA Conversion Algorithm



Equivalent DFA:



Lexical Analyzer Design

1. Convert each of the REs to their respective NFAs.
2. Unify the NFAs.
3. Convert the Unified NFA into a DFA.
4. Examine each final state in the DFA (two cases could arise)
   a) Accepting states are coming from only one NFA.
   b) Accepting states are coming from different NFAs. Use priority rule.

We can use the DFA to do the search over all REs in one shot?

Keyword versus Identifier

- Unify the NFAs for if and Identifier

Have to draw the picture here ☺ But the example is discussed in the class in detail.

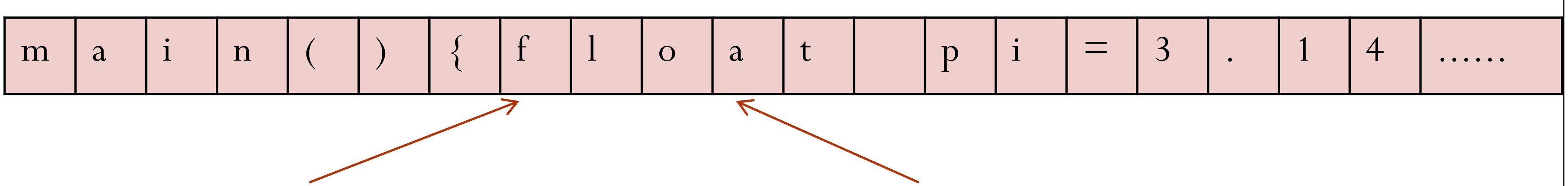
Handling Keywords

Keywords like if, while, boolean, float etc. match the RE specification of Identifiers.

- Distinction between keywords and Identifiers can be taken care of in the DFA
  - Number of states in the DFA increase.
  - Lexical analysis time remains constant
- Recognize keywords as identifiers
  - Store keywords in a Perfect Hash Table.
  - We can check whether an identifier is a keyword in constant time.

Parser and Lexical Analyzer

- Parser doesn't pass a string to the LA and ask what its token type is!
- Parser asks "Hey, give me the next token!"
- Lexical analyzer has to find
  - the extent of the current token and
  - its token type



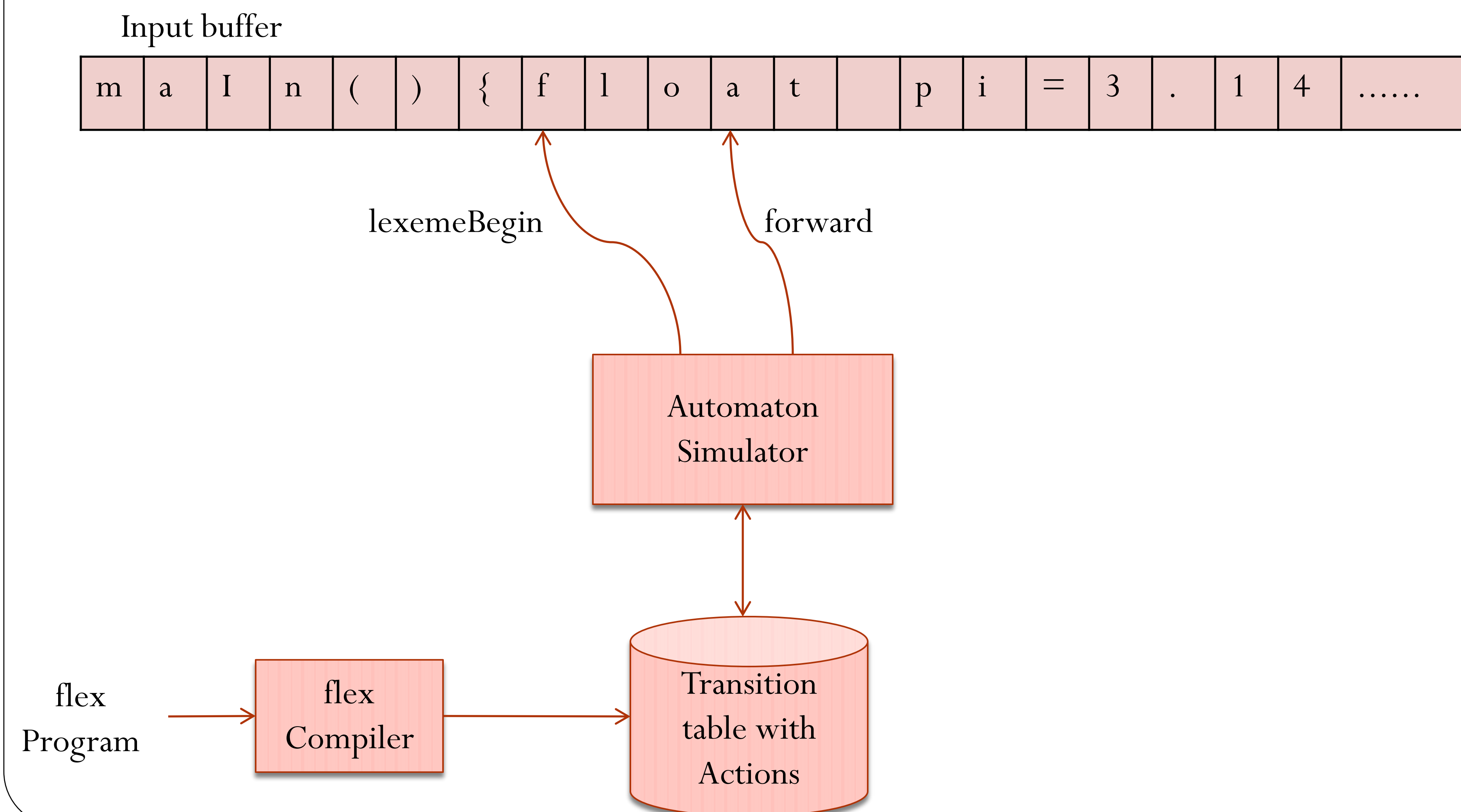
lexemeBegin – Points to the beginning of the current token.

forward – Looks for the extent of the current token.

Scanner Algorithm

```
Scanner( ) {
    while ( *lexemBegin ≠ eof ) do
        state = s0
        forward = lexemeBegin
        while ( *forward ≠ eof  and state ≠ se ) do
            state = nextState(state, *forward)
            forward = forward + 1
        end while
        if ( retract( ) = NULL ) then return NULL  /* retract to the last final state */
        else
            // if multiple tokens are associated with a state,
            // perform the action corresponding to the highest priority token
            perform the action corresponding to the state
        end if
    done
}
```

Structure of Generate Scanner



Complexity of the Scanner Algorithm

- Q1: How much time does it take to extract the next token from the input stream if its length is t?
- Q2: How much time does it take to tokenize an input string of length n?
- Ponder over the two questions with respect to the two regular expressions: ab | (ab)*c

Hard Problems in Lexical Analysis

- Example 1

DO 5 I = 1.25 versus DO 5 I = 1,25

- Example 2

if(x) = 10 versus if(x) h=1

Lexical Analysis Strategies for Tools like grep, find, editors

Should we use the same strategy in grep, find, editors?

Automaton	Initial	Per String
NFA	$O(\|r\|)$	$O(\|r\| \; \|x\|)$
DFA typical case	$O(\|r\|^3)$	$O(\|x\|)$
DFA worst case	$O(\|r\|^2 \, 2^{\|r\|})$	$O(\|x\|)$

- $|x|$ is the length of the input string
- $|r|$ is the length of the regular expression