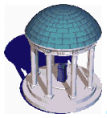


The University of North Carolina at Chapel Hill

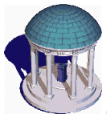
COMP 144 Programming Language Concepts

Spring 2002

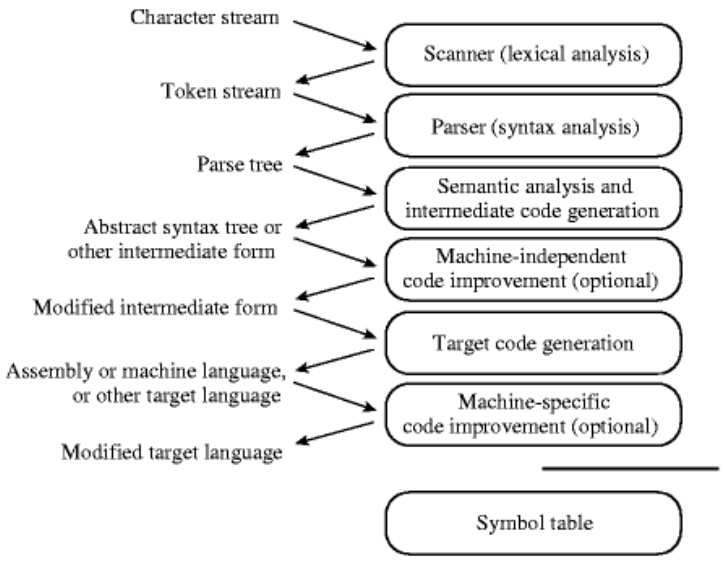
Lecture 3: Lexical Analysis

Felix Hernandez-Campos

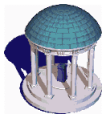
Jan 14



Phases of Compilation

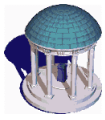


2

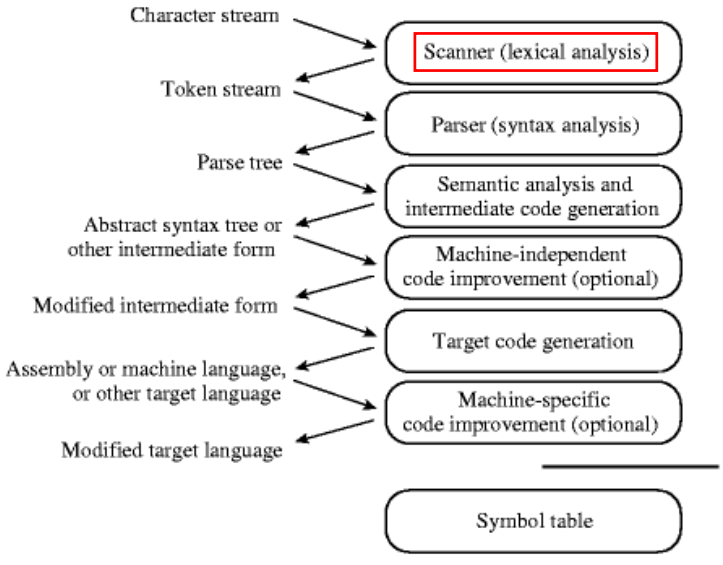


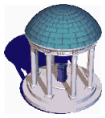
Specification of Programming Languages

- PLs require precise definitions (i.e. no ambiguity)
  - Language *form* (Syntax)
  - Language *meaning* (Semantics)
- Consequently, PLs are specified using formal notation:
  - Formal syntax
    - » Tokens
    - » Grammar
  - Formal semantics



Phases of Compilation

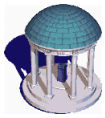




Scanner

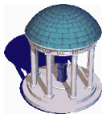
- Main task: identify tokens
  - Basic building blocks of programs
  - *E.g.* keywords, identifiers, numbers, punctuation marks
- Desk calculator language example:

```
read A
sum := A + 3.45e-3
write sum
write sum / 2
```

Formal definition of tokens

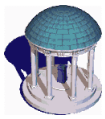
- A set of tokens is a set of strings over an alphabet
  - {read, write, +, -, *, /, :=, 1, 2, …, 10, …, 3.45e-3, …}
- A set of tokens is a *regular set* that can be defined by comprehension using a *regular expression*
- For every regular set, there is a *deterministic finite automaton* (DFA) that can recognize it
  - *i.e.* determine whether a string belongs to the set or not
  - Scanners extract tokens from source code in the same way DFAs determine membership



Regular Expressions

- A regular expression (RE) is:
  - A single character
  - The empty string, ε
  - The concatenation of two regular expressions
    - » *Notation:* RE_1 RE_2 (*i.e.* RE_1 followed by RE_2)
  - The union of two regular expressions
    - » *Notation:* $RE_1 \mid RE_2$
  - The closure of a regular expression
    - » *Notation:* RE*
    - » * is known as the *Kleene star*
    - » * represents the concatenation of 0 or more strings

COMP 144 Programming Language Concepts
Felix Hernandez-Campos
7



Token Definition Example

- Numeric literals in Pascal
  - Definition of the token *unsigned_number*

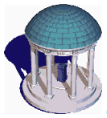
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

unsigned_integer → *digit digit**

unsigned_number → *unsigned_integer* ((. *unsigned_integer*) | ε)
((e (+ | – | ε) *unsigned_integer*) | ε)

- Recursion is not allowed!
- Notice the use of parentheses to avoid ambiguity

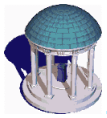
COMP 144 Programming Language Concepts
Felix Hernandez-Campos
8



Scanning

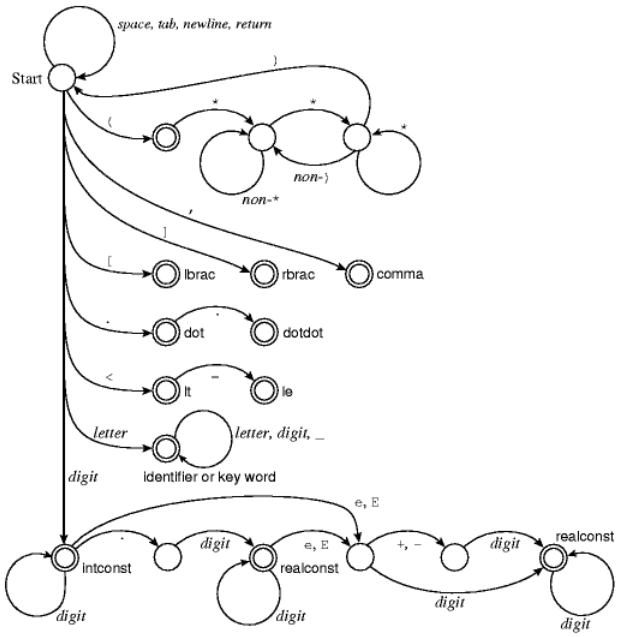
- Pascal scanner Pseudo-code

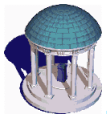
```
we skip any initial white space (spaces, tabs, and newlines)
we read the next character
if it is a ( we look at the next character
     if that is a * we have a comment;
          we skip forward through the terminating *)
     otherwise we return a left parenthesis and reuse the look-ahead
if it is one of the one-character tokens ( [ ] , ; = + - etc.)
     we return that token
if it is a . we look at the next character
     if that is a . we return . .
     otherwise we return . and reuse the look-ahead
if it is a < we look at the next character
     if that is a = we return <=
     otherwise we return < and reuse the look-ahead
etc.
if it is a letter we keep reading letters and digits
     and maybe underscores until we can't anymore;
     then we check to see if it is a keyword
          if so we return the keyword
          otherwise we return an identifier
     in either case we reuse the character beyond the end of the token
if it is a digit we keep reading until we find a non-digit
     if that is not a . we return an integer and reuse the non-digit
     otherwise we keep looking for a real number
          if the character after the . is not a digit we return
          an integer and reuse the . and the look-ahead
etc.
```

DFAs

- Scanners are <u>deterministic finite automata</u> (DFAs)
  - With some *hacks*



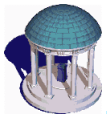


Difficulties

- Keywords and variable names
- Look-ahead
  - Pascal's ranges [1..10]
  - FORTRAN's example

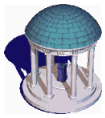
```
DO 5 I=1,25        => Loop 25 times up to label 5

DO 5 I=1.25        => Assign 1.25 to DO5I
```

  - » NASA's Mariner 1 (apocryphal?)
- Pragmas: *significant comments*
  - Compiler options



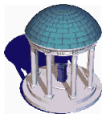
- Outline of the Scanner

```
state := start
loop
    case state of
        start :
            erase text of current token
            case input_char of
            ' ', '\t', '\n', '\r' : no_op
            '[' : state := got_lbrac
            ']' : state := got_rbrac
            ',' : state := got_comma
            ...
            '(' : state := saw_lparen
            '.' : state := saw_dot
            '<' : state := saw_lthan
            ...
            'a'..'z', 'A'..'Z' :
                state := in_ident
            '0'..'9' : state := in_int
            ...
            else error
    ...
    saw_lparen: case input_char of
            '*' : state := in_comment
        else return lparen
    in_comment: case input_char of
            '*' : state := leaving_comment
        else no_op
    leaving_comment: case input_char of
            ')' : state := start
        else state := in_comment
    ...
    saw_dot : case input_char of
            '.' : state := got_dotdot
        else return dot
    ...
    saw_lthan : case input_char of
            '=' : state := got_le
        else return lt
    ...
    in_ident : case input_char of
            'a'..'z', 'A'..'Z', '0'..'9', '_' : no_op
        else
            look up accumulated token
                in keyword table
            if found, return keyword
            else return identifier
    ...
    in_int : case input_char of
            '0'..'9' : no_op
            '.' :
                peek at character beyond input_char;
                if '0'..'9', state := saw_real_dot
                else
                    unread peeked-at character
                    return intconst
            'a'..'z', 'A'..'Z', '_' : error
        else return intconst
    ...
    saw_real_dot : ...
    ...
    got_lbrac : return lbrac
    got_rbrac : return rbrac
    got_comma : return comma
    got_dotdot : return dotdot
    got_le : return le
    ...
append input_char to text of current token
read new input_char
```



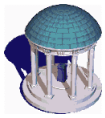
Scanner Generators

- Scanners generators:
  - E.g. lex, flex
  - These programs take a table as their input and return a program (*i.e.* a scanner) that can extract tokens from a stream of characters



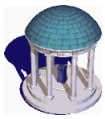
- Table-driven scanner
- Lexical errors

```
state = 1..number of states
action_rec = record
    action : (move, recognize, error)
    new_state : state
    token_found : token

scan_tab : array [char, state] of action_rec
keyword_tab : set of record
    k_image : string
    k_token : token
-- these two tables are created by a scanner generator tool

tok : token
image : string
cur_state : state
cur_char : char

state := start_state
image := null
repeat
    loop
        read cur_char
        case scan_tab[cur_char, cur_state].action
            move:
                cur_state := scan_tab[cur_char, cur_state].new_state
            recognize:
                tok := scan_tab[cur_char, cur_state].token_found
                exit inner loop
            error:
                -- print error message and recover; probably start over
        append cur_char to image
    -- end inner loop
until tok not in [white_space, comment]
look image up in keyword_tab and replace tok with appropriate keyword if found
return (tok, image)
```



Scanners and String Processing

- Scanning is a common task in programming
  - String processing
  - *E.g.* reading configuration files, processing log files,…
- `StringTokenizer` and `StreamTokenizer` in Java
  - http://java.sun.com/products/jdk/1.2/docs/api/java/util/StringTokenizer.html
  - http://java.sun.com/products/jdk/1.2/docs/api/java/io/StreamTokenizer.html
- Regular expressions in Python and other scripting languages



Reading Assignment

- Scott's Chapter 2:
  - Introduction
  - Section 2.1.1
  - Section 2.2.1