How to Create a Programming Language

LECTURE 4: LEXICAL ANALYSIS

1

LECTURE OUTLINE

- **THEORETICAL BACKGROUND**
  - **REGULAR EXPRESSIONS**
  - **CHOMSKY HIERARCHY**
  - **CONTEXT FREE GRAMMARS**
- **SCANNING**

2

PROGRAMMING LANGUAGE SYNTAX

- COMPUTER LANGUAGES MUST BE PRECISE!
  - **SYNTAX:** GRAMMAR
  - **SEMANTICS:** MEANING/INTERPRETATION
  - ABOVE DESCRIPTIONS OF LANGUAGE MUST HAVE NO AMBIGUITY
- SPECIFYING SYNTAX
  - REGULAR EXPRESSIONS
  - CONTEXT FREE GRAMMARS

3

PROGRAMMING LANGUAGE SYNTAX

- EXAMPLE: THE SYNTAX OF ARABIC NUMERALS:
  - A *DIGIT* "IS" : 0 |(OR) 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  - A *NON_ZERO_DIGIT* "IS" : 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  - A *NATURAL_NUMBER* (>0) "IS" A NON_ZERO_DIGIT FOLLOWED BY OTHER DIGITS
  - A NUMBER THAT DOESN'T START WITH 0 CAN BE MATCHED WITH THE REGULAR EXPRESSION *NON_ZERO_DIGIT DIGIT**

4

REGULAR EXPRESSIONS

- A REGULAR EXPRESSION IS:
  - THE EMPTY STRING, DENOTED BY ε
  - A SINGLE CHARACTER
  - TWO REGULAR EXPRESSIONS CONCATENATED
    - *A* IS A REGULAR EXPRESSION, *B* IS A REGULAR EXPRESSION, THEN *AB* IS A REGULAR EXPRESSION
  - ALTERNATION OF TWO REGULAR EXPRESSIONS [SEPARATED BY 'OR' |]
    - GIVEN *A* AND *B* ARE REGULAR EXPRESSIONS, *(A|B)* IS A REGULAR EXPRESSION
  - A REGULAR EXPRESSION FOLLOWED BY KLEENE * (CLOSURE)
    - GIVEN A IS A REGULAR EXPRESSION, *A** IS A REGULAR EXPRESSION

5

REGULAR EXPRESSIONS (EXAMPLE)

- DECIMAL NUMERIC LITERAL
  - digit is *0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9*
  - integer is *digit digit**
  - real is *integer exponent |*
    decimal (exponent | ε)
  - decimal is *digit*(.digit|digit.)digit**
  - exponent is *(e|E)(+|-| ε) integer*
  - number is *(integer | real)*

6

REGULAR EXPRESSIONS

- WORK WELL FOR DEFINING *TOKENS*
- UNABLE TO SPECIFY NESTED CONSTRUCTIONS
  - EXAMPLE: CONTEXT FREE GRAMMAR BNF
    - **EXPR→ ID | NUMBER | -EXPR| (EXPR) | EXPR OP EXPR**
    - **OP → + | -| * | /**
    - (SAME NUMBER OF OPEN AND CLOSE PARENS CANNOT BE DESCRIBED)

7

CHOMSKY HIERARCHY

- Context Free Languages more powerful than RE
- REs are faster to recognize
- Chomsky Hierarchy
  - **Type 3** – Regular Language – Finite Automata
  - **Type 2** – Context Free Languages –Pushdown Automata
  - **Type 1** - Context Sensitive Languages – Turing Machine w/Tape * Input
  - **Type 0** – Unrestricted Language – Turing Machine
    - Type 0/1 too slow for practical use
    - Type 2 – It depends
    - Type 3 – Fast!

8

CONTEXT FREE GRAMMARS (CFG)

- BACKUS-NAUR FORM (BNF) NOTATION:

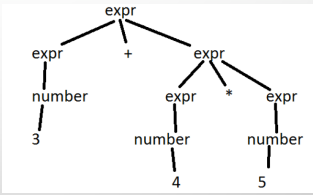
  EXPR ➔ ID | NUMBER | -EXPR | (EXPR) | EXPR OP EXPR

  OP ➔ + | - | * | /
  - EACH 'RULE' IS CALLED A *PRODUCTION*
  - SYMBOLS ON LEFT ARE *NON-TERMINALS*
- A CFG INCLUDES:
  - A SET OF *TERMINALS* T
  - A SET OF *NON-TERMINALS* N
  - A *START SYMBOL* S (NON-TERMINAL)
  - A SET OF *PRODUCTIONS*

9

DERIVATIONS AND PARSE TREES

- A CFG SHOWS HOW A STRING OF TERMINALS IS GENERATED BY A LANGUAGE
  - USING PRODUCTIONS OF A LANGUAGE
  - ➔ SYNTACTICALLY VALID IN THE 'LANGUAGE'
- EX: PARSE TREE FOR 3 + 4 * 5



10

CONTEXT FREE GRAMMARS

- GIVEN EXPRESSION GRAMMAR
  - EXPR := EXPR OP EXPR
  - EXPR := ε
  - OP := + | * | - | /
- PREVIOUS GRAMMAR AMBIGUOUS
  - CAN DERIVE 2 DIFFERENT PARSE TREES!
    - TREE 1: 3 + (4 * 5)
    - TREE 2: (3 + 4) * 5
- BETTER VERSION
  - …INCLUDE PRECEDENCE AND ASSOCIATIVITY

  EXPR ➔ TERM | EXPR ADD_OP TERM

  TERM ➔ FACTOR | TERM MUL_OP FACTOR

  FACTOR ➔ ID | NUMBER | -FACTOR | (EXPR)

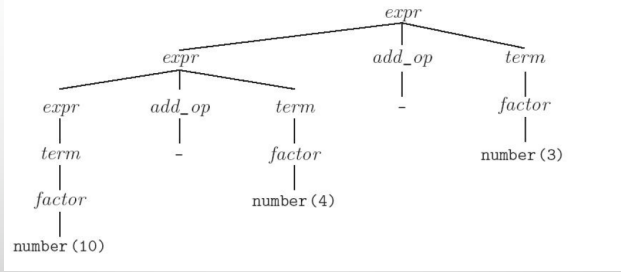
  ADD_OP ➔ + | -

  MULT_OP ➔ * | /

11

CONTEXT FREE GRAMMARS

- PARSE TREE FOR EXPRESSION GRAMMAR (WITH LEFT ASSOCIATIVITY) FOR 10 - 4 - 3



12

SCANNING

- THE (LEXICAL) SCANNER AND PARSER (TOGETHER) RESPONSIBLE FOR DISCOVERING SYNTACTIC STRUCTURE OF PROGRAM
  - THE SCANNER IS RESPONSIBLE FOR
    - TOKENIZING SOURCE
    - REMOVING COMMENTS, (OFTEN) DEALING WITH PRAGMAS (I.E., SIGNIFICANT COMMENTS))
    - SAVING TEXT OF IDENTIFIERS, NUMBERS, STRINGS
    - SAVING SOURCE LOCATIONS (FILE, LINE, COLUMN) FOR ERROR MESSAGES

13

SCANNING

Consider an ad-hoc (hand-written) scanner for a Calculator:

assign → **:=**
plus → **+**
minus → **-**
times → *
div → /
lparen→ **(**
rparen→ **)**
id → ***letter (letter | digit)****
number → ***digit digit****
| ***digit * (. digit | digit .) digit*** *
comment → **/*** ***(non-* | * non-/)* */***
| **//** ***(non-newline)* newline***

14

SCANNING

Read characters one at a time with look-ahead

skip initial white space (spaces, tabs, and newlines)

if cur_char∈{'(', ')', '+', '-', '*'}
  return the corresponding single-character token
  if cur_char= ':'
    read the next character
    if it is '=' then return *assign* else announce an error
if cur_char= '/'
  peek at the next character
  if it is '*' or '/'
    read additional characters until "*/" or
    ***newline* is seen, respectively**
  jump back to top of code
else return *div*

15

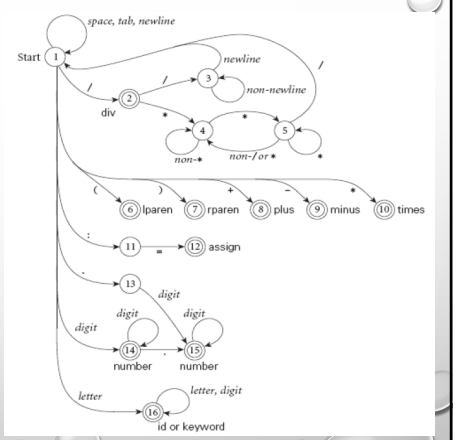
SCANNING

if cur_char = .
  read the next character
  if it is a digit
    read any additional digits
    return number
  else announce an error
if cur_char is a digit
  read any additional digits and at most one decimal point
  return number
if cur_char is a letter
  read any additional letters and digits
  check to see whether the resulting string is *read* or *write*
  if so then return the corresponding token
  else return id
else announce an error

16

SCANNING

- PICTORIAL REPRESENTATION OF A SCANNER FOR CALCULATOR TOKENS, IN THE FORM OF A FINITE AUTOMATON



17

SCANNING

- THIS WAS A DETERMINISTIC FINITE AUTOMATON (DFA)
  - *LEX*, *SCANGEN*, ETC. BUILD THESE THINGS AUTOMATICALLY FROM A SET OF REGULAR EXPRESSIONS
  - SPECIFICALLY, THEY CONSTRUCT A MACHINE THAT ACCEPTS THE LANGUAGE IDENTIFIER | INT CONST | REAL CONST | COMMENT | SYMBOL | ...

18

SCANNING

- RUN THE MACHINE OVER AND OVER TO GET ONE TOKEN AFTER ANOTHER
  - NEARLY UNIVERSAL RULE (**GREEDY SCANNING**):
    - ALWAYS TAKE THE **LONGEST** POSSIBLE TOKEN FROM INPUT THUS *FOOBAR* IS *FOOBAR* AND NEVER *F* OR *FOO* OR *FOOB*
    - ANOTHER EXAMPLE: *3.14159* IS A REAL CONSTANT AND NEVER *3*, *.*, AND *14159*
- REGULAR EXPRESSIONS ***GENERATE*** A REGULAR LANGUAGE
- DFAS ***RECOGNIZE*** IT

19

SCANNING

- THE LEXER TURNS A PROGRAM INTO A STRING OF TOKENS.
  - MATCHES REGULAR EXPRESSIONS TO PROGRAM
  - LEXER CREATES A LIST OF TOKENS
  - TWO SYNTAXES FOR REGULAR EXPRESSIONS: EBNF AND PERL-STYLE REGEX
- SCANNERS TEND TO BE BUILT THREE WAYS
  - WRITING / GENERATING A FINITE AUTOMATON
  - AD-HOC SCANNER CODE (USUALLY NESTED CASE STATEMENTS)
  - TABLE-DRIVEN DFA
- HAND-WRITTEN FINITE AUTOMATON GENERALLY YIELDS FASTEST, MOST COMPACT CODE
  - DOES MANY SPECIAL-PURPOSE THINGS
  - GOOD **AUTOMATICALLY-GENERATED** SCANNERS COME VERY CLOSE

20

SCANNING

- WRITING A PURE DFA AS A SET OF NESTED CASE STATEMENTS IS A SURPRISINGLY USEFUL TECHNIQUE
  - USE PERL, AWK, SED
  - TABLE-DRIVEN DFA IS WHAT LEX AND SCANGEN PRODUCE
    - *LEX(FLEX)* IN THE FORM OF C CODE
    - *SCANGEN* IN THE FORM OF NUMERIC TABLES AND A SEPARATE DRIVER

21

SCANNING

- RULE ABOUT LONGEST-POSSIBLE TOKENS
  - YOU RETURN **ONLY** WHEN THE NEXT CHARACTER CAN'T BE USED TO CONTINUE THE CURRENT TOKEN
  - NEXT CHARACTER GENERALLY NEEDS TO BE SAVED FOR NEXT TOKEN
- MAY NEED TO PEEK AT MORE THAN ONE CHARACTER TO KNOW WHETHER TO PROCEED
  - IN PASCAL, FOR EXAMPLE, WHEN YOU HAVE A 3 AND YOU A SEE A DOT
    - DO YOU PROCEED (IN HOPES OF GETTING 3.14)? OR
    - DO YOU STOP (IN FEAR OF GETTING 3..5)? (DECLARATION OF ARRAYS IN PASCAL, E.G., "ARRAY [1..6] OF INTEGER")

22

PERL-STYLE REGEXP

- Learning by examples:
  - *abcd* - concatenation
  - *a(b|c)d* -grouping
  - *a(b|c)*d* - can apply # of repeats to char or group.
  - *?* = 0-1
  - * = 0-infinity
  - + = 1-infinity
  - *[bc]* -character class – b or c
  - *[a-zA-Z0-9_]* –ranges- any alpha numeric character
  - . -matches any character – one character only
  - *\a* -alpha
  - *\d* -numeric
  - *\w* -word (alpha, num, _)
  - *\s* -whitespace

23

PERL-STYLE REGEXP

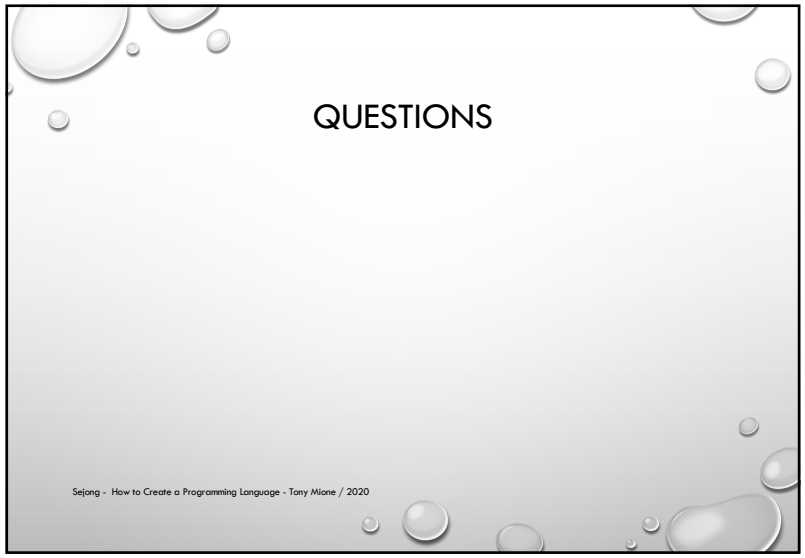
Learning by examples:

How do we write a regexp that matches floats?

digit(.digit|digit.)digit**

\d*(\.\d|\d \.)\d*

24


QUESTIONS
Sejong - How to Create a Programming Language - Tony Mione / 2020

25