Syntax

Regular expression and scanner

(Book 4^{th} Edition) 2.1, 2.2

Introduction

- A designer specifies a language by specifying its syntax and semantics
- To specify syntax, a designer uses
  - Regular expression
  - Context free grammar
- The syntax is used
  - by programmers to understand this language
  - by implementers of language to build a compiler

Specifying syntax

- A program consists of
  - *Tokens*: shortest strings of characters that have individual meaning. E.g., in C, keywords, identifiers, symbols, and constant of various types
  - *Constructs* over *tokens.* Example (given a piece of code) .
- To specify a language, it is sufficient to specify tokens and constructs.
  - Tokens by *regular expressions.*
  - Constructs over tokens by *context free grammars.*

Tokens and regular expressions

- Normally, a number is a basic unit of programs in any language. How to specify what is a number? (a number is a sequence of digits which are of 0, 1, …, 9)

number -> digit digit^

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

->: is, |: or, x^: 0 or more repetition of x,

digit digit^: concatenation of two regular expressions.

- How to specify a “natural number” which is a number the leftmost digit of which may not be 0?
- How to specify a “non negative decimal number”?

Definition of **classical regular expression**

- A character (from a given alphabet) is a *regular expression*
- The empty string, denoted by ε, is a *regular expression*
- The concatenation of two regular expressions is a *regular expression*
- Two regular expressions connected by | is a *regular expression*
- A regular expression followed by ^ (called *Kleene star)* is a *regular expression*

Ways to specify an RE

- Production rules
  - allow ->, |, ^, ε

```
number -> digit digit^
digit -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

- BNF(Backus-Naur Form)
  - allow only ::= (is), | (or)

<number> ::= <digit> | <digit><number>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Scanner

- Scanner (of a regular expression)
  - Input: a character stream (i.e., a program) and a position *p* in this stream
  - Output: output the token starting from *p,* (complains if the regular expression (syntax) is not followed)

- Example of scanner of the following RE for a calculator

  assign -> :=

  plus -> + lparen -> (

  minus -> - rparen ->)

  id -> letter (letter | digit)^ except for read and write

  number -> digit digit^ | ……

  comment -> /* (non-* | *non-/)^ */

- Approach to building a scanner
  – ad hoc algorithm (read Example 2.10 in the book)
  – **Deterministic automata** based approach (systematic)

Automata based scanner: example

- Example – a calculator language (P54) allows tokens like variables (id), assignment, and arithmetic operation +, …
  - RE for tokens of a scientific calculator

assign -> := id -> letter (letter | digit)^

plus -> + lparen -> (

minus -> - rparen ->)

id -> letter (letter | digit)^ *except for* **read** *and* **write**

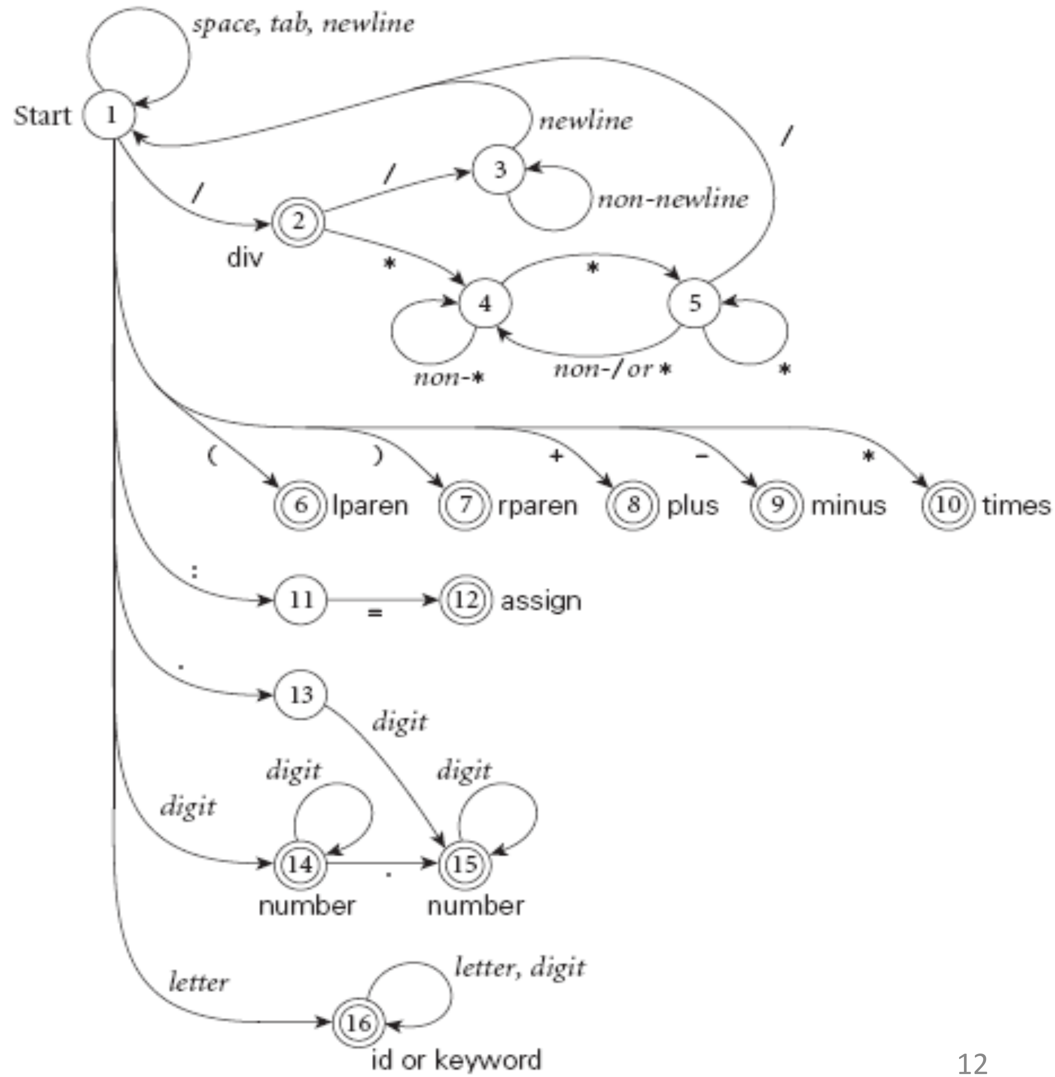
number -> digit digit^ | ……

comment -> /* (non-* | *non-/)^ */

| // (non-newline^) newline

An automata for token recognation

- for the example
  (longest possible token rule: scanning as many letters as possible as long as there is a transition from the current state)



- Example: use the automata to find tokens in

```
2Y – 5*x /* This is a comment */
```

- Recognize a token, using an automata, from an input
  - Know well state, start/final state, transition edge (labeled by a character) from one state to another.
  - To recognize a token from input
    - start from the initial state
    - the input will drive the automata to new states
    - when the automata reaches a final state, and the *next* character in the input can drive the automata to nowhere, a token is recognized.

- Lexical error occurs
  - When the current state is *not* final, and

    the *next* letter in the input stream brings the current state to nowhere, or
  - the current state is not final and there is no letter left in the input stream.

Automata based scanner – general

Given regular expressions

1. Regular expressions -> NFA (non-deterministic automata) [systematic method]
2. NSF -> DFA [systematic method, not discussed in this class]
3. From DFA build a scanner

Regular expressions -> NFA

- Rules for translating an RE to NFA (p58)
  - When the RE is
    - a character c:
      - NFA: [s] c [end]
    - Exp1 Exp2: let Exp1 be [s] <…> [end], Exp2 [s1]… [end]
      - NFA: [s]<…>[s1]…[end]
    - Exp1 | Exp2
      - …
    - Exp^
- Example (P59)

NFA -> DFA

- Skipped

DFA -> scanner

- Scanner algorithms
  – Nested *case* based (example?)
    - state = 1 // start state

      loop

      read cur-char

      case state of // Outer cases: states

      case cur-char of // inner cases

Inner cases: transitions (to new states) or returns a token

– Table (the DFA is represented as a data structure - see section 2.2.3)
  - Note the table in section 2.2.3 extends the DFA earlier to include states 17 and 18. 17 is a final state for "a maximal sequence of white spaces", and 18 is a final state for "comment".
  - Table

State	Space / tab	Newline	Letter	Other (char)
1	17	17	16	-
2	-	-	-	-

  - Other information: which state is a final state and what token type does it represent.