CS5314:Concepts of Programming Languages
Spring 2016

Dr. Barbara G. Ryder
J. Byron Maupin Professor of Engineering
ryder@cs.vt.edu
KWII, Room 2210
http://people.cs.vt.edu/~ryder/

Introduction -1

- Administrivia
- Why study PLs?
- Formal languages, lightly (Ch 1+2, Scott)
  - Backus Naur Form (BNF)
  - Regular expressions and finite state machines
  - Context-free grammars
- Student background questionnaire – Due Thursday, Jan 21st

Info on Course Website

http://people.cs.vt.edu/~ryder/5314/

- *Main* page has course summary, expected work, grading, main topics, and important announcements for everyone in the class
- *Lecture Notes* page will have PDF lecture slides and will suggest relevant textbook sections and auxiliary materials
- *Programming assign*ments page will describe 3 programming assignments and due dates
- *Homework assignments* page will list assigned 'by hand' homeworks and after due date, answers

Course Goals

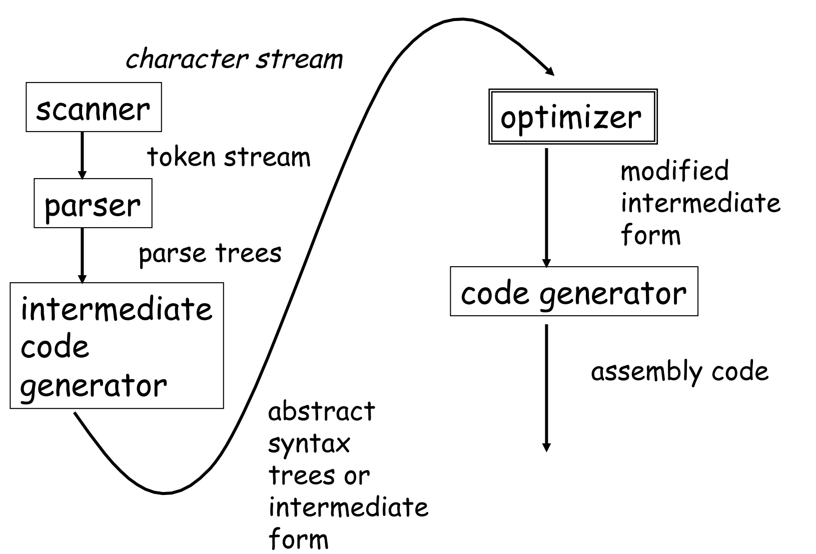
- To make learning new programming languages easier by identifying common features
- To refine understanding of basic structures of programming languages
  - Types, control structures, data objects, naming conventions, and binding etc.
- To study different language paradigms
  - Functional, logic, object-oriented, scripting
  - To ensure an appropriate language is selected for a task

Topics

- PL paradigms: procedural, object-oriented, logic, functional, scripting
- FSAs, RE's, context-free grammars, parsing
- Types: conversion, coercion, equivalence, checking, reconstruction (type-by-use), non-standard types
- Lexical and dynamic scoping
- Lambda calculus and functions as "first class", continuations
- Advanced control flow abstractions and modes of parameter passing
- Data abstraction and specification
- Models of inheritance in object-oriented languages
- Concurrency

Compilation

Scott Ch 1.6



Backus Naur Form (BNF)

- Metasymbols < > ::= |
- Terminal symbols of the PL
  - e.g., keywords, operators
- Nonterminal symbols

<while_stmt> ::= while <expr> do <stmt>

<identifier> ::= <letter> |

<identifier> <digit> | <identifier> <letter>

BNF Examples

- *letter (letter | digit)* *

  <id> ::= <letter> | <id> <letter> | <id> <digit>
- *digit**

  <integer> ::= <integer><digit> | <digit> | ε
- Strings of 1's and 0's where all 1's come before all 0's, that is, 1^*0^*

  <str> ::= <one> <zero>

  <one> ::= 1 <one> | 1 | ε

  <zero> ::= 0 <zero> | 0 | ε
- If statement

  <if_stmt> ::= if <expr> then <statement> else <statement>

Extended BNF (EBNF)

- Nonterminals begin with capital letters or are shown in a different font
- {…} means repeat the enclosed 0 or more times
- […] means the enclosed is optional
- (…) is used for grouping, usually with the alternation symbol |
- If { }, [ ], or ( ) are terminals in the PL being defined, then when they are used as terminals they must be underlined

  <u>{</u> <u>}</u> terminals, { } metasymbols

EBNF Examples

Identifier ::= Letter { LetterorDigit }

LetterorDigit ::= Letter | Digit

Expr ::= [Expr -] Subexpr

IfStmt ::= if LogicExpr then Stmt [else Stmt]

CompoundStmt ::= begin Stmt {; Stmt} end

WhileStmt ::= while <u>(</u> LogicExpr <u>)</u> Stmt {; Stmt}

ArrayElement ::= Identifier <u>[</u> Identifier <u>]</u>

…

Formally

- A PL is a set of strings, called *sentences*, over some finite alphabet of symbols, called *terminals*
  - Not necessarily a finite set
- Rules describe how to combine the terminals into well-formed sentences in the PL - syntax
- PL constructs are categorized by the complexity of their descriptive rules
- *Regular expressions* used to describe tokens (atomic bits) of PLs
  - e.g., identifiers, numerical constants, keywords
  - Defined recursively
  - Recognized by a finite state automaton (FSA)

Regular Expressions

PL construct	RE Notation	Language
	an empty RE	{ }
symbol a	a	{a}
null symbol	ε	{ε}
R,S regular exprs	R | S	$L_R \cup L_S$
a,b terminals	*a|b (alternation)*	*{a,b}*
R,S regular exprs	RS	$L_R L_S$
a,b terminals	*ab (concatenation)*	*{ab}*

Regular Expressions

PL construct	RE Notation	Language
R,S regular exprs	R^*	$\{\varepsilon\} \cup L_R \cup L_R L_R \cup L_R L_R L_R \ldots$
a	a^*	$\{\varepsilon, a, aa, aaa, \ldots\}$
R,S regular exprs	R^+	$L_R \cup L_R L_R \cup L_R L_R L_R \ldots$
a	a^+	$\{a, aa, aaa, \ldots\}$

Note: $\varepsilon\, a = a\, \varepsilon = a$

Precedence is {* +} ----concatenation ---- |

high to low

(all are left associative operators)

RE Examples

1 | 2	{1,2}
1^* | 2	{2, ε,1,11,111,...}
1 2^*	{1, 12, 122, 1222, ...}
1 2^* | 0^+	{0,00,000,...,1,12,122,...}
$(1 \mid 2)^*$	{ε,1,2,12,11,21,22,...}
$(0 \mid 1)^*$ 1	Binary numbers that end in 1

RE's for PLs

- Let *letter* stand for a|b|c|…|z and *digit* stand for 0|1|2|3|4|5|6|7|8|9
  - *letter (letter | digit)* * is identifier
  - *digit* + is an integer constant
  - *digit* * . *digit* + is real number
- Which identifiers are described by
  - *letter (letter | digit)* * ?

  ABC 0C B% X1
- Which of the following are legal real numbers described by
  - *digit* * . *digit* + ? .5 1.5 2 4. 6.3 0.2

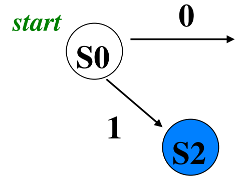
Formal Language Theory

- Offers a way to describe computation problems formulated as language recognition problems and prove their difficulty
- Recognizers for languages are more complex as the languages become more complex
  - Simple constructs correspond to FSAs
    - Keywords, numerical constants
  - More complex constructs correspond to Push-down Automata
    - If statements, looping statements, declarations
  - Even more complex constructs correspond to more complex automata
    - Type checking of use with declared type

Finite State Automaton (FSA)

- Recognizer of the language generated by a regular expression
- Described by
  - <set of states, labeled transitions, start state, final state(s)>

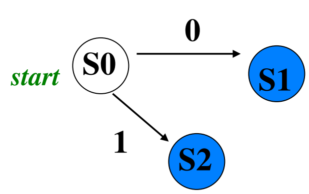
<{S0,S1,S2}, S0 --0--> S1, S0 --1--> S2, S0, {S1,S2}>



Recognizes the language {1, 0}

FSA

- FSA *accepts* or *recognizes* an input string iff there is a path from its start state to a final state such that the labels on the path are the terminals in that string
  - Empty transitions signify illegal moves; can think of FSA going to a sink error state

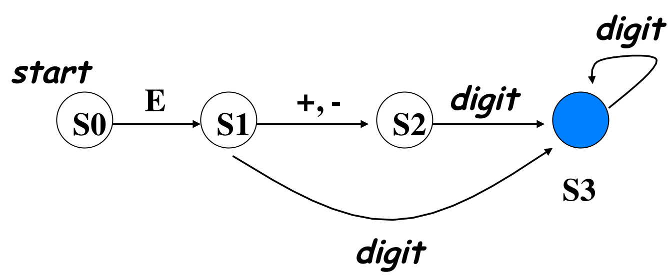


states:	inputs: 0	1
S0	S1	S2
S1	---	---
S2	---	---

transition table

Example

Exponent in scientific notation:
E (+ | -) *digit* $^+$ | E *digit* $^+$



Grammar

- A formalism to describe the sentences of a PL
- <set of terminals, set of non-terminals, productions, special symbol>
  - Terminals are alphabet symbols (e.g., +)
  - Non-terminals represent PL constructs (e.g., Stmt)
  - Productions are rules for forming syntactically correct constructs
  - Special symbol tells where to start applying the rules

Example

<letter>::=
a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

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

<identifier> ::= <letter> | <identifier> <letter> | <identifier> <digit>

<0-assign_stmt> ::= <identifier> = 0;

//nonterminals are {<letter>,<digit>,<0-assign_stmt>, <identifier>}

//special symbol is <0-assign_stmt>

Regular PLs

- Describe the simple constructs in real PLs
- Form of rules
  - Each righthandside is length <= 2 symbols
    - A terminal or non-terminal
    - A non-terminal followed by a terminal
- All PLs describable by REs can be written as regular grammars

  e.g., $1\ 2^* \mid 0^+$ N::= X | Y
  X ::= 1 | X 2
  Y ::= 0 | Y 0