Chapter 2 :: Programming Language Syntax

Programming Language Pragmatics

Michael L. Scott

Introduction

- programming languages need to be precise
  - natural languages less so
  - both form (syntax) and meaning (semantics) must be unambiguous
  - we need good notation (or a metalanguage) to describe precise languages by recognizing tokens
    - regular expressions
    - context-free grammars

Tokens

- tokens are the building blocks of programs
  - shortest strings with linguistic meaning
    - similar to parts of speech in natural language
  - examples
    - keywords (e.g., **`for`**, **`if`**, **`else`** in Python)
    - identifiers (e.g., names of variables and functions)
    - symbols (e.g., mathematical operators **`+`** and **`*`**)
    - literals (e.g., integer **`37`**, floating point **`3.14159`**)

Scanning

- scanning, or lexical analysis, is the first step in making sense of a computer program
- scanner reads a stream of characters and groups them into tokens (or identifies them as invalid)
  - tokenization
  - e.g., a Python scanner identifies keywords such as **`def`** and identifiers such as **`foo`** and **`bar`**
  - considerations
    - case sensitivity
    - international characters
    - maximum lengths

Scanning

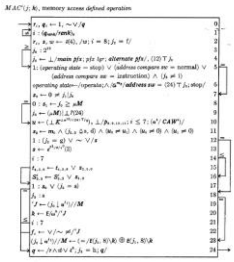
- considerations
  - case sensitivity
    - Python, C – case-sensitive
    - Fortran – case-insensitive
    - Haskell – names beginning with capital letters have special meaning
  - character set
    - Python – allows Unicode letters in identifiers
    - C, Fortran – must be ASCII
    - Haskell – allows characters such as single quotes in identifiers
  - length limits
    - C – only first 63 characters guaranteed to matter
    - Fortran – identifiers are ≤ 6 characters

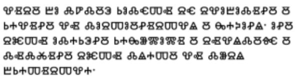
Regular Expressions

- the lexical form of a token is typically specified by a regular expression
  - also called a regex or a regexp
- describe patterns of characters
  - we are interested in three characteristics:
    1. valid characters which comprise the string
    2. smallest string(s)
    3. special pattern(s) of strings produced (must fully capture description of strings)
  - for example, x[xyz]*
    1. strings comprised of x, y, and z
    2. smallest string: x
    3. strings must begin with x
- help us find tokens in the programming language
- useful in unix/linux environments

Regular Expressions

- given an alphabet Σ, a regular expression (RE) describes a set of strings composed of characters from Σ
- alphabet
  - non-empty set of characters
  - examples
    - ASCII character set
    - EBCDIC character set
    - APL's freaky-deaky character set
    - Unicode Glagolitic character set
  - for concreteness, think ASCII
- a string is a sequence of characters





Regular Expressions Formal Definition

- regular expressions over an alphabet Σ are defined as follows
  1. the empty set ∅
  2. ε denoting the set consisting of only the empty string (e.g., "" in Python)
  3. if c ∈ Σ, then c is an RE denoting the set that contains only the character c
  4. α|β denoting the union of regular expressions α and β
  5. αβ denoting the set of concatenations of strings of regular expressions α and β
  6. α∗ denoting the union of the concatenations of α with itself, zero or more times
     - α∗ is called the Kleene closure of α (for Stephen Cole Kleene, 1909–1994, American mathematician)

Regular Expressions Operators

- if Σ = {a, b, c, d}, then
  - the RE *a* denotes the set containing only *a*
  - the RE *a*|*b* represents the set {*a*, *b*} (*a* or *b*)
  - the RE *ab* represents the set {*ab*}
  - the RE (*a*|*b*)*c* represents the set {*ac*, *bc*}
  - the RE (*a*|*b*)(*c*|*d*) represents the set {*ac*, *ad*, *bc*, *bd*}
  - the RE *a*∗ represents the set {ε, *a*, *aa*, *aaa*, *aaaa*, . . .}

RE Operator Precedence

- operator precedence for purposes of association

  (α)
  α∗
  αβ
  α|β

- thus, *ab***c*|*d* is interpreted as ((*a*(*b**))*c*)|*d*, which represents the {*d*, *ac*, *abc*, *abbc*, *abbbc*, . . .}

RE Notational Conventions

- in keeping with the convention for REs in many computer systems, we delimit sets of characters with square brackets [] and omit any separators (requiring we suspend concatenation inside square brackets to avoid ambiguity)
  - [*abc*] denotes the set {*a*, *b*, *c*}
  - *abc* represents the set {*abc*}
- inside square brackets we denote a range of characters with the first and last character connected by a dash - (this assumes the alphabet is ordered)
  - [0-9] represents the set of decimal digits (0|1|2|3|4|5|6|7|8|9)
  - [a-zA-Z] denotes all of the lowercase and uppercase letters

RE Examples

- REs for tokens in Python
  1. an identifier (e.g., variable name) must begin with either an underscore or an alphabetic character followed by zero or more underscores or alphanumeric characters
     - assuming it's ASCII, can write this in RE form as
       [_a-zA-Z][_0-9a-zA-Z]*
  2. a positive integer can be described as a nonzero decimal digit followed by zero or more decimal digits w
     - we can capture this succinctly with the RE
       [1-9][0-9]*
  3. in decimal notation, a positive floating point number is a decimal digit followed by a decimal point followed by zero or more decimal digits, or a decimal point followed by one or more decimal digits
     - an appropriate RE is
       [0-9].[0-9]* | .[0-9][0-9]*

RE Examples

4. A positive floating point number in scientific notation is more complicated, as any of the forms

   100.0, 1e2, 1e+2, 1.e2, 1.0e2, 1.0e02, 0.01, 1e-2, 1.e-2, 1.0e-2, 1.0e-02

   plus any of these preceded by 0 are valid, captured by the RE

   [0-9]*(.[0-9]* | .[0-9]*[eE][+-ε][0-9][0-9]* | [eE][+-ε][0-9][0-9]*)

   This RE means
   - zero or more digits, followed by
     - a decimal point followed by zero or more digits
     - a decimal point followed by zero or more digits then e or E
     - e or E

   If there is an e or E, it is followed by
   - an optional sign - or +, followed by
     - a decimal digit, followed by
     - zero or more decimal digits

13

13

REs in Practice

- REs are extremely useful when specifying textual patterns for searches
- languages such as C and Python, POSIX operating systems (i.e., Unix and Linux), and editors such as emacs and vim include tools for specifying and searching for regular expressions
- most of these facilities can express more than just regular expressions and include “syntactic sugar” to make it easier to specify patterns

14

14

RE Metacharacters

- in many RE systems (e.g., C, Python, POSIX), the following characters are metacharacters

  \ ^ $. | ? * + ( ) [ ]
  - these have special meaning other than their literal meaning

15

15

Regular Expressions in POSIX

- the POSIX standard specifies two types of Res
  - Basic Regular Expressions (BREs)
  - Extended Regular Expressions (EREs)
  - (there is actually a third type, Simple Regular Expressions, but their use is discouraged)
- having two kinds of REs is problematic
  - in BREs, metacharacters used as metacharacters (rather than as literals) are escaped with \
  - in EREs, metacharacters used as literals (rather than as metacharacters) are escaped with \ (default)
  - some tools (e.g., egrep) expect EREs by default; others (e.g., grep) expect BREs but use EREs if the -E option is specified

16

16

POSIX ERE Metacharacters

( ) matches the enclosed RE
- () matches the null string

. matches any single character
- d.g matches "dog "
- [d.g] matches only a 'd', '.', or 'g'

^ matches the start of the string
- ^dog matches the "dog" in "dogleg" but not in "underdog"

$ matches the end of string

* matches 0 or more repetitions of the preceding RE
- ab* will match "a", "ab", or "abb", "abbb", ….

+ matches one or more repetitions of the preceding RE
- ab+ will not match "a" but will match "ab", "abb", "abbb", ….

? matches zero or one repetitions of the preceding RE
- ab? will match either "a" or "ab"
- useful when a character is optional in the RE

17

17

POSIX ERE Metacharacters (cont.)

[ ] a bracket expression is used to indicate a list of characters
- the list can contain characters
  - [dog] will match 'd', 'o', or 'g'
  - use \] to include a literal ’]’ inside a bracketed list
- ranges of characters indicated by two characters separated by a –
  - [a-z] matches any lowercase ASCII letter (includes ~ in EBCDIC!)
  - [1-5][0-9] will match any two-digit number from 10 to 59
  - if - is escaped (e.g., [a\-z]) it will match a literal -
- most metacharacters lose their special meaning inside sets
  - [(+*)] will match any of '(', '+', '*', or ')'
- if the first character in a bracketed list is ^ then all the characters that are not in the set will be matched
  - [^42] will match any character except '4' and '2'
  - the caret ^ has no special meaning if not the first character

18

18

POSIX ERE Metacharacters (cont.)

|	specifies a match to either the RE that precedes or follows the |
{m}	matches the preceding RE exactly m times
{,n}	matches the preceding RE not more than n times
{m,}	matches the preceding RE at least m times
{m,n}	matches the preceding RE between m and n times

19

19

POSIX ERE Character Classes

shorthand for common sets of characters

Shorthand	ASCII RE	Characters Represented
[:upper:]	[A-Z]	uppercase letters
[:lower:]	[a-z]	lowercase letters
[:alpha:]	[a-zA-Z]	all letters
[:alnum:]	[a-zA-Z0-9]	all digits and letters
[:digit:]	[0-9]	digits
[:xdigit:]	[0-9A-Fa-f]	hexadecimal digits
[:punct:]	[.,!?:;]	punctuation
[:blank:]	[\t]	space and tab characters
[:space:]	[\t\n\r\f\v]	whitespace characters
[:cntrl:]		control characters
[:graph:]	[^\t \n\r\f\v]	printed characters
[:print:]	[^\t\n\r\f\v]	printed characters and space

20

20

Regular Expressions in Python

- Python's re module provides tools for handling regular expressions
  - similar to POSIX, but has more features and can handle patterns more complex than REs
  - HOWTO guide is a good place to start

21

21

Finite Automata and REs

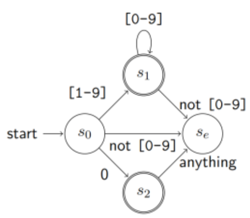
- regular expressions can be recognized using finite automata (FA)
  - also called finite state machines (CSCI 423)
- a finite automaton consumes a string, one character at a time
  - depending on the character, the FA may or may not change to a new state
- the FA accepts (recognizes) a string if and only if the FA finds itself in one of a distinguished set of final (or accepting) states when the entire string has been consumed
- a coin-operated vending machine is a physical example of an FA

22

22

Finite Automata Example

- an unsigned integer is either zero, or one or more digits where the first digit is nonzero
  - 0|[1-9][0-9]*
- here is an FA that will recognize the RE; final states are indicated by double circles



23

23

Finite Automata Formalism

- preceding example described an FA in terms of its transition diagram
- formally, an FA is a quintuple $(S, \Sigma, \delta, s_0, SF)$, where
  1. S is the set of states, which is finite
  2. Σ is the alphabet used by the recognizer, typically the union of the edge labels in the transition diagram; Σ must be finite
  3. $\delta(s, c) : S \times \Sigma \rightarrow S$ is a function of a state $s \in S$ and a character $c \in \Sigma$
     1. encodes the transitions of the FA
     2. when the FA is in state s and sees a c, it makes a transition to the state δ(s, c)
  4. $s_0 \in S$ is the designated start state
  5. $SF \subset S$ is the set of final states
- the cost of applying an FA to a string is proportional to the length of the string, even if the FA has a large number of states

24

24

Kleene's Equivalence Theorem

- Kleene showed that REs and FAs were equivalent in the sense that
  - given an RE, you can build an FA that will recognize that RE, and
  - given an FA, you can build an RE that is recognized by the FA
- in fact, there exist practical algorithms for transforming an RE into an FA and an FA into an RE
- the ability to transform REs into FAs that recognize them makes possible to automate the generation of scanners!

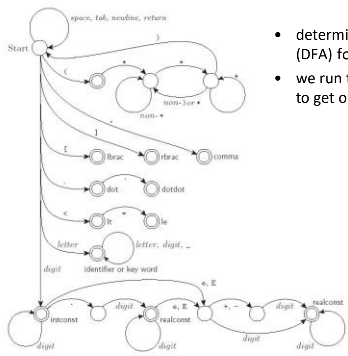
lex and flex

- scanners (also called lexers or lexical analyzers) can be automatically generated
- one of the earliest scanner generators in widespread use was lex, developed at Bell Labs in the 1970s
- an open source analog, flex, was developed in the 1980s and is available from the GNU project
- a scanner generator takes as its input the names of the tokens and the REs that describe them (as well as actions to take when a token is recognized) and generates code that implements the scanner

Scanning

- recall that the scanner is responsible for
  - tokenizing source
  - removing comments
  - saving text of identifiers, numbers, strings
- suppose we are building an ad-hoc (hand-written) scanner for Pascal
  - we read the characters one at a time with look-ahead
  - always take the longest possible token from the input
- regular expressions "generate" a regular language; DFAs "recognize" it

Scanning



- deterministic finite automaton (DFA) for recognizing Pascal tokens
- we run the machine over and over to get one token after another

Parsing

- a parser is responsible for recognizing syntax
- scanners and parsers typically work together
  - the scanner feeds the parser a stream of tokens
  - the parser analyzes the tokens for grammatically correct statements

Context-free Grammars

- a context-free grammar (CFG) *G* is a set of rules that describe what strings of symbols are valid sentences in a language
- the collection of sentences that can be derived from G is called the language defined by *G*, denoted *L(G)*
- the notation for CFGs is sometimes called Backus-Naur Form (BNF)
- with Kleene star and other facilitating symbols, the notation is termed Extended BNF (EBNF)

Context-free Grammars

- consider the following CFG *S*

  SheepNoise → **baa** *SheepNoise* | **baa**

- meaning
  - *SheepNoise* can derive the word baa followed by more *SheepNoise*
  - *SheepNoise* can derive the word **baa**
- grammar S describes the language
  baa, baa baa, baa baa baa, …

Context-free Grammars

- in general a CFG consists of
  - nonterminal symbols (e.g., *SheepNoise*)
    - appear on left-hand side
  - terminal symbols (e.g., **baa**)
    - appear only on right-hand side
    - words in the language
  - productions (e.g., single statement in CFG *S*)
    - statements with arrows showing possible replacements
  - start symbol (e.g., *SheepNoise*)
    - nonterminal
    - if not explicitly stated, the left-hand non-terminal of the first production

Derivations

- CFG *S*

  SheepNoise → **baa** *SheepNoise* | **baa**

- we can derive strings, such as **baa baa**
  - begin with the start symbol, *SheepNoise*
  - choose a grammar rule for replacement
    - only one per line
  - repeat until the string consists of only terminals
  - strings of intermediate nonterminal/terminal strings called sentential forms

SheepNoise → **baa** *SheepNoise*
→ **baa baa**

Backus-Naur Form

- the notation for context-free grammars (CFG) is sometimes called Backus-Naur Form (BNF)
  - names for John Backus (who developed Fortran) and Peter Naur
  - necessary since regular expressions cannot specify nested constructs
  - used to define the syntax of a language
- written in their original notation, the sheep grammar *S* is

  〈SheepNoise〉 ::= baa 〈SheepNoise〉
  | baa

- we will use
  - → instead of ::=
  - italics for nonterminals
  - Courier font for terminals

Extended Backus-Naur Form

- for convenience, Extended Backus-Naur Form (EBNF) is often used
- same as BNF, but augmented with extra operators
  - optional list: choose one or none [ ]
  - choose one from list ( )
  - choose zero or more instances { }
  - choose zero or more instances *
  - choose one or more instances +
- these symbols should never appear in any derivation
  - instead, make all decisions for operators in one step
  - for example,
    - production: *str* → **x***
    - derivation: *str* → **x x x**

Formal Definition of CFG

- in a context-free grammar, production rules allow only a single nonterminal on the left-hand side
- in a context-sensitive grammar, production rules allow multiple nonterminals on the left-hand side

Formal Definition of CFG

- the formal definition of *S* is

  T = {**baa**}
  N = {*SheepNoise*}
  s = *SheepNoise*
  P = {*SheepNoise* → **baa** *SheepNoise*, *SheepNoise* → **baa**}

Example

- consider the grammar

  Integer → *Digit* | *Integer Digit*
  Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

- we can derive any unsigned integer, like 352, from this grammar

  Integer → *Integer Digit*
  → *Integer* 2
  → *Integer Digit* 2
  → *Integer* 5 2
  → *Digit* 5 2
  → 3 5 2

Example

- a different derivation of 352

  Integer → *Integer Digit*
  → *Integer Digit Digit*
  → *Digit Digit Digit*
  → 3 *Digit Digit*
  → 3 5 *Digit*
  → 3 5 2

- this is called a leftmost derivation since at each step, the leftmost nonterminal is replaced
- the previous derivation was a rightmost derivation

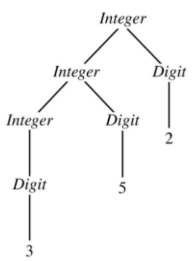
Parse Tree

- a parse tree is a graphical representation of a derivation
  - each internal node of the tree corresponds to a step in the derivation
  - the children of a node represent a right-hand side of a production
  - each leaf node represents a symbol of the derived string reading from left to right

Parse Tree

- parse tree for leftmost derivation of 352

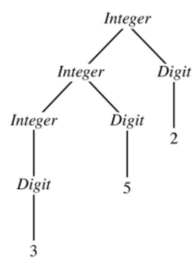
  Integer → *Integer Digit*
  → *Integer Digit Digit*
  → *Digit Digit Digit*
  → 3 *Digit Digit*
  → 3 5 *Digit*
  → 3 5 2



Parse Tree

- parse tree for rightmost derivation of 352

  Integer → *Integer Digit*
  → *Integer* 2
  → *Integer Digit* 2
  → *Integer* 5 2
  → *Digit* 5 2
  → 3 5 2



- parse tree is the same as for the left derivation

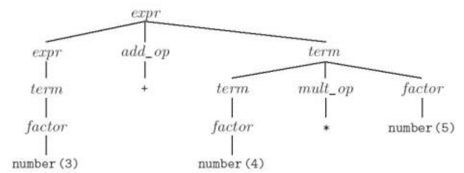
Expression Grammars

- expression grammar with precedence and associativity

1. *expr* ⟶ *term* | *expr add_op term*
2. *term* ⟶ *factor* | *term mult_op factor*
3. *factor* ⟶ `id` | `number` | `-` *factor* | `(` *expr* `)`
4. *add_op* ⟶ `+` | `-`
5. *mult_op* ⟶ `*` | `/`

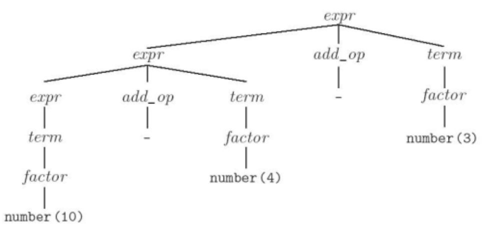
Expression Grammars

- parse tree for expression grammar (with precedence) for **3 + 4 * 5**



Expression Grammars

- parse tree for expression grammar (with left associativity) for **10 - 4 - 3**



Expression Grammars

- another grammar with precedence and associativity
  - + and – are left-associative operators in mathematics
  - * and / have higher precedence than + and –
- Grammar G_1

Expr -> Expr + Term | Expr – Term | Term
*Term -> Term * Factor | Term / Factor |*
Term % Factor | Factor
*Factor -> Primary ** Factor | Primary*
Primary -> 0 | ... | 9 | (Expr)

Expression Grammars

- parse tree for 4**2**3 + 5 * 6 + 7

Expression Grammars

- associativity and precedence shown in the structure of the parse tree
  - highest precedence at the bottom
  - left-associativity on the left at each level

Precedence	Associativity	Operators
3	right	**
2	left	* / %
1	left	+ -

Ambiguous Grammars

- a grammar is ambiguous if one of its strings has two or more different parse trees
  - grammar G_1 above is unambiguous
- ambiguous expression grammar G_2 equivalent to G_1

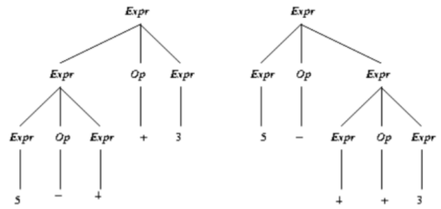
  Expr -> *Expr Op Expr* | (*Expr*) | *Integer*
  Op -> + | - | * | / | % | **

  - fewer productions and nonterminals, but ambiguous

49

Ambiguous Grammars

- ambiguous parse of 5 – 4 + 3 using G_2



50

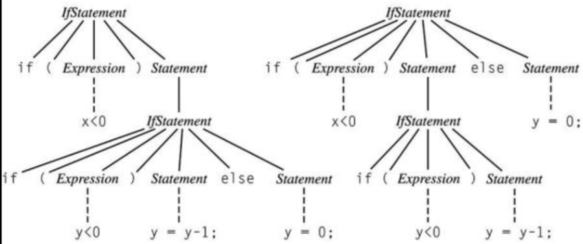
Dangling Else Ambiguity

- with which if statement does the else associate?

```
if (x < 0)
    if (y < 0)  y = y  - 1;
    else y = 0;
```

51

Dangling Else Ambiguity



52

Dangling Else Solutions

- Python
  - uses indentation to specify nesting level
- C and C++
  - associate each else with closest if
  - use { } or begin/end to override
- other languages
  - use explicit delimiter to end every conditional (e.g., if..fi)

53

Scanning and Parsing Programs

- GCD program in C

```
void main ()  {
   int i, j;

   scanf ("%d %d", &i, &j);

   while (i != j)
      if (i > j) i = i - j;
      else j = j - i;

   printf ("%d\n", i);
}
```

54

Scanning and Parsing Programs

- tokens

```
void     scanf     while     i       printf
main     (         (         =       (
(        "%d %d"   i         i       "%d\n"
)        ,         !=        -       ,
{        &         j         j       i
int      i         )         ;       )
i        ,         if        else    ;
,        &         (         j       }
j        j         i         =
;        )         >         j
         ;         j         -
                   )         i
```

55

55

Token Categories

- identifier
  - function name, variable name
  - main, x
- keyword
  - type names, control structures
  - int, if, for, while, return, etc.
- literal
  - constants
  - 3.14, “hello”, ‘c’
- operator
  - mathematical, specialized
  - +, =, sizeof
- delimiter
  - (,), {,], ;

56

56

Tokens and Categories

- tokens and their categories
  - must be shown for every token, in order
  - only first two columns shown below; need to show all

```
void     keyword      scanf     identifier
main     identifier   (         delimiter
)        delimiter    "%d %d"   literal
{        delimiter    ,         delimiter
int      keyword      &         operator
i        identifier   i         identifier
,        delimiter    ,         delimiter
j        identifier   &         operator
;        delimiter    j         identifier
                      )         delimiter
                      ;         delimiter

etc.
```

57

57