

6.035

Fall 2006

Specifying Languages with Regular Expressions and Context-Free Grammars

Saman Amarasinghe

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology

Outline

- Review of Lexical Analysis
- Regular Expressions
- Finite State Automata
- Regular Expressions to Automata
- NFA vs DFA
- Lexical Analysis continued
- Context Free Grammar
- Ambiguous Grammars


Language Definition Problem

- How to precisely define language
- Layered structure of language definition
 - Start with a set of letters in language
 - Lexical structure - identifies "words" in language (each word is a sequence of letters)
 - Syntactic structure - identifies "sentences" in language (each sentence is a sequence of words)
 - Semantics - meaning of program (specifies what result should be for each input)
 - Today's topic: lexical and syntactic structures

Specifying Formal Languages

- Huge Triumph of Computer Science
 - Beautiful Theoretical Results
 - Practical Techniques and Applications
- Two Dual Notions
 - Generative approach (grammar or regular expression)
 - Recognition approach (automaton)
- Many theorems/algorithms about converting one approach automatically to another

What is Lexical Analysis?

- Source program text  Tokens
- Example of Tokens
 - Operators = + - > ({ := == <>
 - Keywords if while for int double
 - Numeric literals 43 6.035 -3.6e10 0x13F3A
 - Character literals 'a' '\w' '\v'
 - String literals "6.891" "Fall 98" "\\\" = empty"
- Example of non-tokens
 - White space space(' ') tab('\t') end-of-line('\n')
 - Comments /*this is not a token*/

A Lexical Analyzer in Action

```
for ID("var1") eq_op Num(10) ID("var1") leq_op
```

- Partition input program text into subsequence of characters corresponding to tokens
- Attach the corresponding attributes to the tokens
- Eliminate white space and comments

Lexical Analyzer needs to...

- Precisely identify the type of token that matches the input string
 - 6036 Num(6035)
 - X6035 ID("X6035")
- Precisely describe different types of tokens
 - FORTRAN DO I=1,10
 - C++ for(int i=1; I<= 10; I++)
 - C-shell foreach i (1 2 3 4 5 6 7 8 9 10)
- Use *regular expressions* to precisely describe what strings each type of token can recognize

Simon Arnsperth 7 6.035 ©MIT Fall 2006

Specifying Lexical Structure Using Regular Expressions

- Have some alphabet Σ = set of letters
- Regular expressions are built from:
 - ϵ - empty string
 - Any letter from alphabet Σ
 - r_1r_2 - regular expression r_1 followed by r_2 (sequence)
 - $r_1 | r_2$ - either regular expression r_1 or r_2 (choice)
 - r^* - iterated sequence and choice $\epsilon | r | rr | \dots$
 - Parentheses to indicate grouping/precedence

Simon Arnsperth 8 6.035 ©MIT Fall 2006

Concept of Regular Expression Generating a String

Rewrite regular expression until have only a sequence of letters (string) left

General Rules

- $r_1 | r_2 \rightarrow r_1$
- $r_1 | r_2 \rightarrow r_2$
- $r^* \rightarrow rr^*$
- $r^* \rightarrow \epsilon$

Example

$(0 | 1)^*. (0 | 1)^*$
 $(0 | 1)(0 | 1)^*. (0 | 1)^*$
 $1(0 | 1)^*. (0 | 1)^*$
 $1. (0 | 1)^*$
 $1. (0 | 1)(0 | 1)^*$
 $1. (0 | 1)$
 1.0

Simon Arnsperth 9 6.035 ©MIT Fall 2006

Nondeterminism in Generation

- Rewriting is similar to equational reasoning
- But different rule applications may yield different final results

Example 1

$(0 | 1)^*. (0 | 1)^*$
 $(0 | 1)(0 | 1)^*. (0 | 1)^*$
 $1(0 | 1)^*. (0 | 1)^*$
 $1. (0 | 1)^*$
 $1. (0 | 1)(0 | 1)^*$
 $1. (0 | 1)$
 1.0

Example 2

$(0 | 1)^*. (0 | 1)^*$
 $(0 | 1)(0 | 1)^*. (0 | 1)^*$
 $0(0 | 1)^*. (0 | 1)^*$
 $0. (0 | 1)^*$
 $0. (0 | 1)(0 | 1)^*$
 $0. (0 | 1)$
 0.1

Simon Arnsperth 10 6.035 ©MIT Fall 2006

Concept of Language Generated by Regular Expressions

- Set of all strings generated by a regular expression is language of regular expression
- In general, language may be (countably) infinite
- String in language is often called a token

Simon Arnsperth 11 6.035 ©MIT Fall 2006

Examples of Languages and Regular Expressions

- $\Sigma = \{ 0, 1, . \}$
 - $(0 | 1)^*. (0 | 1)^*$ - Binary floating point numbers
 - $(00)^*$ - even-length all-zero strings
 - $1^*(01^*01^*)^*$ - strings with even number of zeros
- $\Sigma = \{ a, b, c, 0, 1, 2 \}$
 - $(a | b | c)(a | b | c | 0 | 1 | 2)^*$ - alphanumeric identifiers
 - $(0 | 1 | 2)^*$ - trinary numbers

Simon Arnsperth 12 6.035 ©MIT Fall 2006

Match and Create the Regular Expressions

- | | |
|--------------------------|-----------|
| 1. $0(0 1)^*0$ | a. 000000 |
| 2. $((\epsilon 0)1^*)^*$ | b. 01010 |
| 3. $((0 1)0(0 1))^*$ | c. 010101 |
| | d. 101010 |
| | e. 001100 |

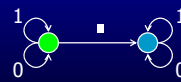
- All strings of 0's and 1's that does not contain the substring 011

Simon Arnsperth 12 6.035 ©MIT Fall 2004

Alternate Abstraction Finite-State Automata

- Alphabet Σ
- Set of states with initial and accept states
- Transitions between states labeled with letters

$(0|1)^*.0(0|1)^*$



● Start state

● Accept state

Simon Arnsperth 13 6.035 ©MIT Fall 2004

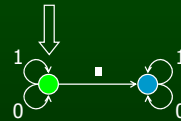
Automaton Accepting String

- Conceptually, run string through automaton
 - Have current state and current letter in string
 - Start with start state and first letter in string
 - At each step, match current letter against a transition whose label is same as letter
 - Continue until reach end of string or match fails
 - If end in accept state, automaton accepts string
- Language of automaton is set of strings it accepts

Simon Arnsperth 14 6.035 ©MIT Fall 2004

Example

Current state



● Start state

● Accept state

1 1 . 0

Current letter

Simon Arnsperth 15 6.035 ©MIT Fall 2004

Generative Versus Recognition

- Regular expressions give you a way to generate all strings in language
- Automata give you a way to recognize if a specific string is in language
 - Philosophically very different
 - Theoretically equivalent (for regular expressions and automata)
- Standard approach
 - Use regular expressions when define language
 - Translated automatically into automata for implementation

Simon Arnsperth 16 6.035 ©MIT Fall 2004

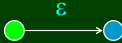
From Regular Expressions to Automata

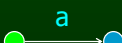
- Construction by structural induction
- Given an arbitrary regular expression r
- Assume can convert r to an automaton with
 - One start state
 - One accept state
- Show how to convert all constructors to deliver an automaton with
 - One start state
 - One accept state

Simon Arnsperth 17 6.035 ©MIT Fall 2004

Basic Constructs

● Start state
● Accept state

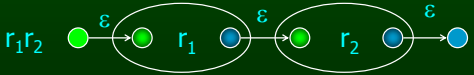
ϵ 

$a \in \Sigma$ 

6.035 ©MIT Fall 2005

Sequence

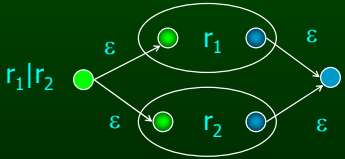
● Old start state ● Start state
● Old accept state ● Accept state



6.035 ©MIT Fall 2005

Choice

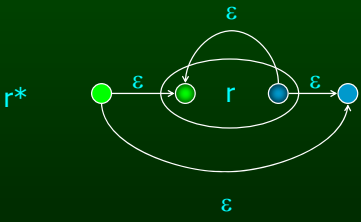
● Old start state ● Start state
● Old accept state ● Accept state



6.035 ©MIT Fall 2005

Kleene Star

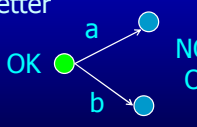
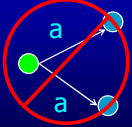
● Old start state ● Start state
● Old accept state ● Accept state



6.035 ©MIT Fall 2005

NFA vs. DFA

- DFA
 - No ϵ transitions
 - At most one transition from each state for each letter
- NFA – neither restriction

OK 
 NOT OK 

6.035 ©MIT Fall 2005

Conversions

- Our regular expression to automata conversion produces an NFA
- Would like to have a DFA to make recognition algorithm simpler
- Can convert from NFA to DFA (but DFA may be exponentially larger than NFA)
 - Simple algorithm available (check a text book for details)

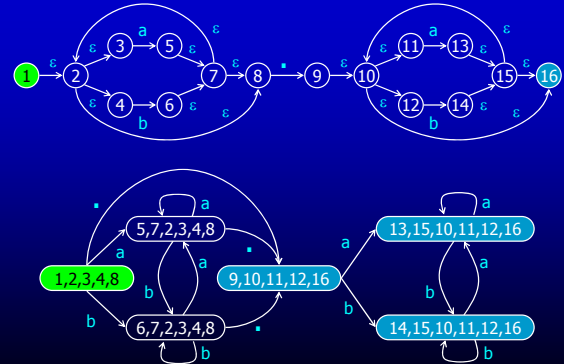
6.035 ©MIT Fall 2005

NFA to DFA Construction

- DFA has a state for each subset of states in NFA
 - DFA start state corresponds to set of states reachable by following ϵ transitions from NFA start state
 - DFA state is an accept state if an NFA accept state is in its set of NFA states
- To compute the transition for a given DFA state D and letter a
 - Set S to empty set
 - Find the set N of D's NFA states
 - For all NFA states n in N
 - Compute set of states N' that the NFA may be in after matching a
 - Set S to S union N'
 - If S is nonempty, there is a transition for a from D to the DFA state that has the set S of NFA states
 - Otherwise, there is no transition for a from D

Source: Adapted from [1]. 75 6.035 ©MIT Fall 2005

NFA to DFA Example for $(a|b)^*. (a|b)^*$



Source: Adapted from [1]. 76 6.035 ©MIT Fall 2005

Lexical Structure in Languages

- Each language typically has several categories of words. In a typical programming language:
 - Keywords (if, while)
 - Arithmetic Operations (+, -, *, /)
 - Integer numbers (1, 2, 45, 67)
 - Floating point numbers (1.0, .2, 3.337)
 - Identifiers (abc, i, j, ab345)
- Typically have a lexical category for each keyword and/or each category
- Each lexical category defined by regexp

Source: Adapted from [1]. 77 6.035 ©MIT Fall 2005

Lexical Categories Example

- IfKeyword = if
- WhileKeyword = while
- Operator = +|-|*|/
- Integer = [0-9] [0-9]*
- Float = [0-9]*. [0-9]*
- Identifier = [a-z]([a-z]|[0-9])*
- Note that $[0-9] = (0|1|2|3|4|5|6|7|8|9)$
 $[a-z] = (a|b|c|\dots|y|z)$
- Will use lexical categories in next level

Source: Adapted from [1]. 78 6.035 ©MIT Fall 2005

Write a Regular Expression

- All strings of the wedge alphabet { <, > }
 -
- Strings with open wedges followed by close wedges
 -
- Strings with matching wedges
 -

Source: Adapted from [1]. 79 6.035 ©MIT Fall 2005

Programming Language Syntax

- Regular languages insufficient for specifying programming language syntax
- Why? Constructs with nested syntax
 - $(a+(b-c))*(d-(x-(y-z)))$
 - if $(x < y)$ if $(y < z)$ a = 5 else a = 6 else a = 7
- Regular languages lack state required to model nesting
- Canonical example: nested expressions
- No regular expression for language of parenthesized expressions

Source: Adapted from [1]. 80 6.035 ©MIT Fall 2005

Context-Free Grammar

- Set of terminals
 $\{ Op, Int, Open, Close \}$
 Each terminal defined by regular expression
- Set of nonterminals
 $\{ Start, Expr \}$
- Set of productions
 - Single nonterminal on LHS
 - Sequence of terminals and nonterminals on RHS

$Op = +|-|*|/$
 $Int = [0-9] [0-9]^*$
 $Open = <$
 $Close = >$
 $Start \rightarrow Expr$
 $Expr \rightarrow Expr Op Expr$
 $Expr \rightarrow Int$
 $Expr \rightarrow Open Expr Close$

Language Generation

- have a current list
- start with *Start* nonterminal as the current list
- loop until no more nonterminals in the list
 - choose a nonterminal in current list
 - choose a production with that nonterminal in LHS
 - replace nonterminal with RHS of production
- for each terminal in the list
 - substitute the regular expressions
 - generated string

Note: different choices produce different strings

Sample Derivation

$Op = +|-|*|/$
 $Int = [0-9] [0-9]^*$
 $Open = <$
 $Close = >$

$Start$
 $Expr$
 $Expr Op Expr$
 $Open Expr Close Op Expr$
 $Open Expr Op Expr Close Op Expr$
 $Open Int Op Expr Close Op Expr$
 $Open Int Op Expr Close Op Int$
 $Open Int Op Int Close Op Int$

- $Start \rightarrow Expr$
- $Expr \rightarrow Expr Op Expr$
- $Expr \rightarrow Int$
- $Expr \rightarrow Open Expr Close$

Sample Derivation

$Op = +|-|*|/$
 $Int = [0-9] [0-9]^*$
 $Open = <$
 $Close = >$

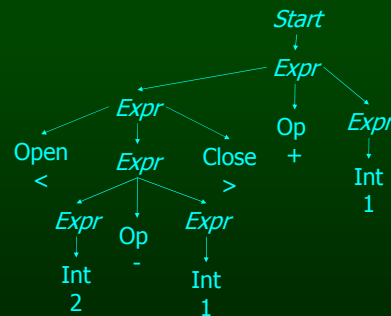
$Open Int Op Int Close Op Int$
 $< Int Op Int Close Op Int$
 $< [0-9][0-9]^* Op Int Close Op Int$
 $< 2 Op Int Close Op Int$
 $< 2 +|-|*|/ Int Close Op Int$
 $< 2 - Int Close Op Int$
 $< 2 - [0-9][0-9]^* Close Op Int$
 $< 2 - 1 Close Op Int$
 $< 2 - 1 > Op Int$
 $< 2 - 1 > +|-|*|/ Int$
 $< 2 - 1 > + Int$
 $< 2 - 1 > + [0-9][0-9]^*$
 $< 2 - 1 > + 1$

- $Start \rightarrow Expr$
- $Expr \rightarrow Expr Op Expr$
- $Expr \rightarrow Int$
- $Expr \rightarrow Open Expr Close$

Parse Tree

- Internal Nodes: Nonterminals
- Leaves: Terminals
- Edges:
 - From Nonterminal of LHS of production
 - To Nodes from RHS of production
- Captures derivation of string

Parse Tree for $<2-1>+1$



Language Recognition

Op = +|-|*|/
 Int = [0-9] [0-9]*
 Open = <
 Close = >

1) $Start \rightarrow Expr$
 2) $Expr \rightarrow Expr Op Expr$
 3) $Expr \rightarrow Int$
 4) $Expr \rightarrow Open Expr Close$

< 2 - 1 > + 1
 Open 2 - 1 > + 1
 Open Int - 1 > + 1
 Open Int Op 1 > + 1
 Open Int Op Int > + 1
 Open Int Op Int Close + 1
 Open Int Op Int Close Op 1
 Open Int Op Int Close Op Int

Language Recognition

Op = +|-|*|/
 Int = [0-9] [0-9]*
 Open = <
 Close = >

1) $Start \rightarrow Expr$
 2) $Expr \rightarrow Expr Op Expr$
 3) $Expr \rightarrow Int$
 4) $Expr \rightarrow Open Expr Close$

Open Int Op Int Close Op Int
 Open Expr Op Int Close Op Int
 Open Expr Op Expr Close Op Int
 Open Expr Close Op Int
 Expr Op Int
 Expr Op Expr
 Expr
 Start

What is this Grammar?

What is the language of the following grammar?
 $Start \rightarrow S$
 $S \rightarrow (L)$
 $S \rightarrow a$
 $L \rightarrow L, S$
 $L \rightarrow S$

Write a parse tree for
 (a, (a, a))

Ambiguity in Grammar

- Grammar is ambiguous if there are multiple derivations (therefore multiple parse trees) for a single string
- Derivation and parse tree usually reflect semantics of the program
- Ambiguity in grammar often reflects ambiguity in semantics of language (which is considered undesirable)

Ambiguity Example

Two parse trees for 2-1+1

Tree corresponding to <2-1>+1

```

graph TD
  Start --> Expr1[Expr]
  Expr1 --> Expr2[Expr]
  Expr1 --> Op1[Op]
  Expr1 --> Expr3[Expr]
  Expr2 --> Expr4[Expr]
  Expr2 --> Op2[Op]
  Expr2 --> Int1[Int]
  Expr4 --> Int2[Int]
  Expr4 --> Op3[Op]
  Expr4 --> Int3[Int]
  Int2 --> Int2_val[2]
  Op3 --> Op3_val[-]
  Int3 --> Int3_val[1]
  Op1 --> Op1_val[+]
  Int1 --> Int1_val[1]
          
```

Tree corresponding to 2-<1+1>

```

graph TD
  Start --> Expr1[Expr]
  Expr1 --> Expr2[Expr]
  Expr1 --> Op1[Op]
  Expr1 --> Expr3[Expr]
  Expr2 --> Int2[Int]
  Int2 --> Int2_val[2]
  Op1 --> Op1_val[-]
  Expr3 --> Expr4[Expr]
  Expr3 --> Op2[Op]
  Expr3 --> Expr5[Expr]
  Expr4 --> Int1[Int]
  Int1 --> Int1_val[1]
  Op2 --> Op2_val[+]
  Expr5 --> Int3[Int]
  Int3 --> Int3_val[1]
          
```

Eliminating Ambiguity

Solution: hack the grammar

<p>Original Grammar</p> <p>$Start \rightarrow Expr$ $Expr \rightarrow Expr Op Expr$ $Expr \rightarrow Int$ $Expr \rightarrow Open Expr Close$</p>	<p>Hacked Grammar</p> <p>$Start \rightarrow Expr$ $Expr \rightarrow Expr Op Int$ $Expr \rightarrow Int$ $Expr \rightarrow Open Expr Close$</p>
--	---

Conceptually, makes all operators associate to left

Parse Trees for Hacked Grammar

Only one parse tree for 2-1+1!

Valid parse tree

No longer valid parse tree

©MIT Fall 2004

Precedence Violations

- All operators associate to left
- Violates precedence of * over +
- 2-3*4 associates like <2-3>*4

Parse tree for 2-3*4

©MIT Fall 2004

Hacking Around Precedence

<p>Original Grammar</p> <p>Op = + - * /</p> <p>Int = [0-9] [0-9]*</p> <p>Open = <</p> <p>Close = ></p> <p>Start → Expr</p> <p>Expr → Expr Op Int</p> <p>Expr → Int</p> <p>Expr → Open Expr Close</p>	<p>Hacked Grammar</p> <p>AddOp = + -</p> <p>MulOp = * /</p> <p>Int = [0-9] [0-9]*</p> <p>Open = <</p> <p>Close = ></p> <p>Start → Expr</p> <p>Expr → Expr AddOp Term</p> <p>Expr → Term</p> <p>Term → Term MulOp Num</p> <p>Term → Num</p> <p>Num → Int</p> <p>Num → Open Expr Close</p>
--	--

©MIT Fall 2004

Parse Tree Changes

Old parse tree for 2-3*4

New parse tree for 2-3*4

©MIT Fall 2004

General Idea

- Group Operators into Precedence Levels
 - * and / are at top level, bind strongest
 - + and - are at next level, bind next strongest
- Nonterminal for each Precedence Level
 - Term is nonterminal for * and /
 - Expr is nonterminal for + and -
- Can make operators left or right associative within each level
- Generalizes for arbitrary levels of precedence

©MIT Fall 2004

What is different?

<p>Hacked Grammar I</p> <p>AddOp = + -</p> <p>MulOp = * /</p> <p>Int = [0-9] [0-9]*</p> <p>Open = <</p> <p>Close = ></p> <p>Start → Expr</p> <p>Expr → Expr AddOp Term</p> <p>Expr → Term</p> <p>Term → Term MulOp Num</p> <p>Term → Num</p> <p>Num → Int</p> <p>Num → Open Expr Close</p>	<p>Hacked Grammar II</p> <p>AddOp = + -</p> <p>MulOp = * /</p> <p>Int = [0-9] [0-9]*</p> <p>Open = <</p> <p>Close = ></p> <p>Start → Expr</p> <p>Expr → Expr AddOp Term</p> <p>Expr → Term</p> <p>Term → Open Expr Close</p> <p>Term → Term MulOp Num</p> <p>Term → Num</p> <p>Num → Int</p>
--	--

©MIT Fall 2004

Parser

- Converts program into a parse tree
- Can be written by hand
- Or produced automatically by parser generator
 - Accepts a grammar as input
 - Produces a parser as output
- Practical problem
 - Parse tree for hacked grammar is complicated
 - Would like to start with more intuitive parse tree

Solution

- Abstract versus Concrete Syntax
 - Abstract syntax corresponds to “intuitive” way of thinking of structure of program
 - Omits details like superfluous keywords that are there to make the language unambiguous
 - Abstract syntax may be ambiguous
 - Concrete Syntax corresponds to full grammar used to parse the language
- Parsers are often written to produce abstract syntax trees.

Abstract Syntax Trees

- Start with intuitive but ambiguous grammar
- Hack grammar to make it unambiguous
 - Concrete parse trees
 - Less intuitive
- Convert concrete parse trees to abstract syntax trees
 - Correspond to intuitive grammar for language
 - Simpler for program to manipulate

Example

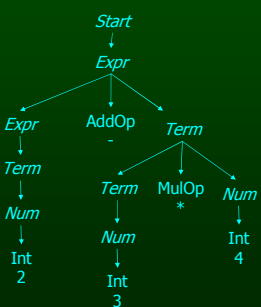
Hacked Unambiguous Grammar

AddOp = +|-
 MulOp = *|/
 Int = [0-9] [0-9]*
 Open = <
 Close = >
 Start → Expr
 Expr → Expr AddOp Term
 Expr → Term
 Term → Term MulOp Num
 Term → Num
 Num → Int
 Num → Open Expr Close

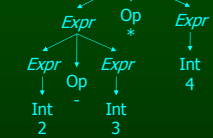
Intuitive but Ambiguous Grammar

Op = *|/|+|-
 Int = [0-9] [0-9]*
 Start → Expr
 Expr → Expr Op Expr
 Expr → Int

Concrete parse tree for <2-3>*4

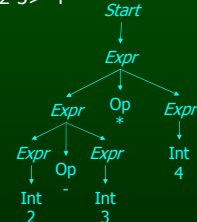


Abstract syntax tree for <2-3>*4

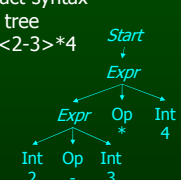


- Uses intuitive grammar
- Eliminates superfluous terminals
 - Open
 - Close

Abstract parse tree for <2-3>*4



Further simplified abstract syntax tree for <2-3>*4



Summary

- Lexical and Syntactic Levels of Structure
 - Lexical – regular expressions and automata
 - Syntactic – grammars
- Grammar ambiguities
 - Hacked grammars
 - Abstract syntax trees
- Generation versus Recognition Approaches
 - Generation more convenient for specification
 - Recognition required in implementation