CS 536 Announcements for Thursday, February 6, 2025

Programming Assignment 2

- has been released
- due Tuesday, February 18

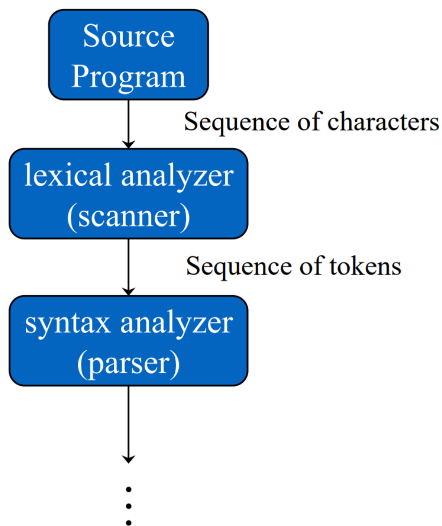
Last Time

- non-determinisitic FSMs
- equivalence of NFAs and DFAs
- regular expressions
- regular languages
- regular expressions → DFAs
- language recognition → tokenizers
- scanner generators
- JLex

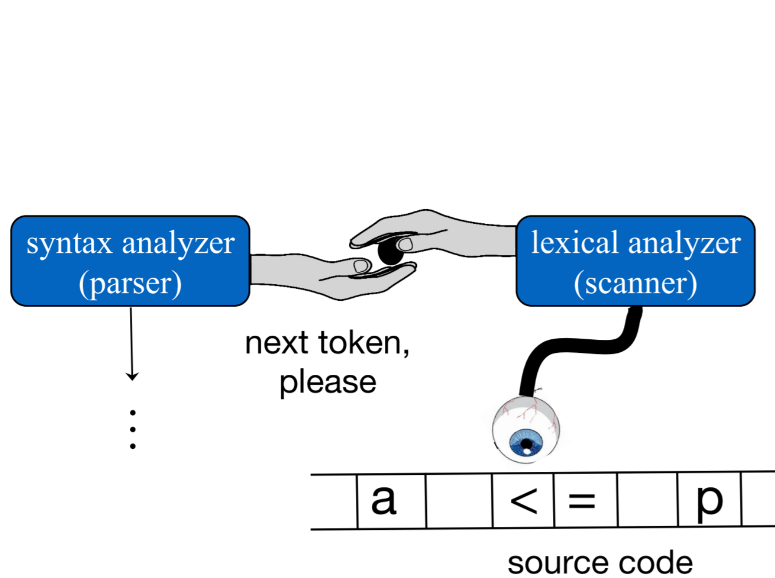
Today

- CFGs
- Makefiles
- resolving ambiguity
- expression grammars
- list grammars

Recall big picture



Conceptual organization



Why regular expressions are not good enough

Regular expression wrap-up

\+ perfect for tokenizing a language

– limitations

- define only limited family of languages
  - can't be used to specify all the programming constructs we need
- no notion of structure

Regexs cannot handle "matching"

Example: $L_{()} = \{ (^n)^n \text{ where } n > 0\}$

Theorem: No regex/DFA can describe the language $L_{()}$

Proof by contradiction: Suppose there exists a DFA A for $L_{()}$ where A has N states.

Then A has to accept the string $(^N)^N$ with some sequence of states

By the pigeonhole principle, there exists $i, j \leq N$ where $i < j$ such that

So

In other words,

No notion of structure

Consider the following stream of tokens: ID ASSIGN ID PLUS ID

The Chomsky Language Hierarchy

Language class:

recursively enumerable

context-sensitive

context-free

regular

Context-free grammar (CFG)

= a set of recursive rewriting rules to generate patterns of strings

Formal definition: A CFG is a 4-tuple (N, ∑, P, S)

- N = set of **non-terminals**
- ∑ = set of **terminals**
- P = set of **productions**
- S = initial non-terminal symbol ("start symbol"), S ϵ N

Productions

Production syntax : LHS → RHS

Language defined by a CFG

= set of strings (i.e., sequences of terminals) that can be derived from the start non-terminal

To derive a string (of terminal symbols):

- set Curr_Seq to start symbol
- repeat
  - find a non-terminal x in Curr_Seq
  - find production of the form x → α
  - "apply" production: create new Curr_Seq by replacing x with α
- until Curr_Seq contains no non-terminals

Derivation notation

- derives
- derives in one or more steps
- derives in zero or more steps

L(G) = language defined by CFG G

=

Example grammar

Terminals

BEGIN

END

SEMICOLON

ASSIGN

ID

PLUS

Non-terminals

prog

stmts

stmt

expr

Productions

1) prog → BEGIN stmts END

2) stmts → stmts SEMICOLON stmt
3) | stmt

4) stmt → ID ASSIGN expr

5) expr → ID
6) | expr PLUS ID

Example derivation

Productions

1) prog → BEGIN stmts END
2) stmts → stmts SEMICOLON stmt
3) | stmt
4) stmt → ID ASSIGN expr
5) expr → ID
6) | expr PLUS ID

Derivation

prog

$\Longrightarrow$ BEGIN stmts END

$\Longrightarrow$ BEGIN stmts SEMICOLON stmt END

$\Longrightarrow$ BEGIN stmt SEMICOLON stmt END

$\Longrightarrow$ BEGIN ID ASSIGN expr SEMICOLON stmt END

$\Longrightarrow$ BEGIN ID ASSIGN expr SEMICOLON ID ASSIGN expr END

$\Longrightarrow$ BEGIN ID ASSIGN ID SEMICOLON ID ASSIGN expr END

$\Longrightarrow$ BEGIN ID ASSIGN ID SEMICOLON ID ASSIGN expr PLUS ID END

$\Longrightarrow$ BEGIN ID ASSIGN ID SEMICOLON ID ASSIGN ID PLUS ID END

Parse trees

= way to visualize a derivation

To derive a string (of terminal symbols):

- set root of parse tree to start symbol
- repeat
  - find a leaf non-terminal x
  - find production of the form x → α
  - "apply" production: symbols in α become the children of x
- until there are no more leaf non-terminals

Derived sequence determined from leaves, from left to right

Productions

1) prog → BEGIN stmts END

2) stmts → stmts SEMICONOLON stmt
3) | stmt

4) stmt → ID ASSIGN expr

5) expr → ID
6) | expr PLUS ID

Makefiles

Basic structure

```
<target>: <dependency list>
     <command to satisfy target)
```

Example

```
Example.class: Example.java IO.class
     javac Example.java

IO.class: IO.java
     javac IO.java
```

Make creates an internal **dependency graph**

- a file is rebuilt if one of its dependencies changes

Variables – for common configuration values to use throughout your makefile

Example

```
JC = /s/std/bin/javac
JFLAGS = -g

Example.class: Example.java IO.class
     $(JC) $(JFLAGS) Example.java

IO.class: IO.java
     $(JC) $(JFLAGS) IO.java
```

Phony targets

- target with no dependencies
- use make to run commands:

Example

```
clean:
     rm -f *.class
```

Programming Assignment 2

Modify:

- bach.jlex
- P2.java
- Makefile

Makefile

```
###
# testing - add more here to run your tester and compare
# its results to expected results
###
test:
	java -cp $(CP) P2
	diff allTokens.in allTokens.out

###
# clean up
###

clean:
	rm -f *~ *.class bach.jlex.java

cleantest:
	rm -f allTokens.out
```

Running the tester

```
vm-instunix-07(53)% make test
java -cp ./deps:. P2
3:1 ****ERROR**** ignoring illegal character: a
diff allTokens.in allTokens.out
3d2
< a
make: *** [Makefile:40: test] Error 1
```

CFG review

formal definition: CFG $G = (N, \Sigma, P, S)$

CFG generates a string by applying
productions until no non-terminals remain

$\Rightarrow^+$ means "derives in 1 or more steps"

language defined by a CFG *G*

  $L(G) = \{ w \mid s \Rightarrow^+ w \}$ where
  s = start is the start non-terminal of G, an
  w = sequence consisting of (only) terminal symbols or ε

Derivation order

1) prog → BEGIN stmts END

2) stmts → stmts SEMICOLON stmt
3) | stmt

4) stmt → ID ASSIGN expr

5) expr → ID
6) | expr PLUS ID

Leftmost derivation :

Rightmost derivation :

Expression Grammar Example

1) expr → INTLIT
2) | expr PLUS expr
3) | expr TIMES expr
4) | LPAREN expr RPAREN

Derive: 4 + 7 * 3

For grammar G and string w, G is **ambiguous** if there is

OR

OR

Grammars for expressions

Goal: write a grammar that correctly reflects precedences and associativities

Precedence

- use different non-terminal for each precedence level
- start by re-writing production for lowest precedence operator first

Example

1) expr → INTLIT
2) | expr PLUS expr
3) | expr TIMES expr
4) | LPAREN expr RPAREN

Grammars for expressions (cont.)

What about associativity? Consider 1 + 2 + 3

Definition: recursion in grammars

A grammar is ***recursive*** **in non-terminal *x*** if
$x \Rightarrow^+ \alpha\, x\, \gamma$ for non-empty strings of symbols α and γ

A grammar is ***left-recursive*** **in non-terminal *x***
if $x \Rightarrow^+ x\, \gamma$ for non-empty string of symbols γ

A grammar is ***right-recursive*** **in non-terminal *x*** if
$x \Rightarrow^+ \alpha\, x$ for non-empty string of symbols α

In expression grammars

for left associativity, use left recursion

for right associativity, use right recursion

Example

Extend this grammar to add exponentiation (POW)

Add exponentiation (POW) to this grammar, with the correct precedence and associativity.

```
expr   →  expr PLUS term
       |  term
term   →  term TIMES factor
       |  factor
factor →  INTLIT
       |  LPAREN expr RPAREN
```

List grammars

Example a list with no separators, e.g., A B C D E F G

Another ambiguous example

stmt → IF cond THEN stmt
| IF cond THEN stmt ELSE stmt
| . . .

Given this word in this grammar: `if a then if b then s1 else s2`
How would you derive it?