Analyzing Text Files

Today’s topic: extracting information from large data files
- pattern matching (e.g., grep) for simple cases
- regular expressions in Perl
- background on languages and grammars
- CFG parsing when Perl REs don’t suffice

Goal: Information for Project #2 (feature parser)

Pattern Matching

- To gather information based on small, well-known attributes simple pattern matching should work
- Example: how many genes are in an annotation?
  ```
  % grep gene NC_000913.gbk
  ```

Pattern Matching (cont’d)

- Problem -- there are too many hits:
  ```
  gene complement(114522..115724)
  /gene="hofC"
  /gene="hofC"
  N-ter as; alternate gene name hofC"
  gene complement(115714..117099)
  ```
- We want only the first and last lines from this section

Regular Expressions

- Use regular expressions to narrow the search
- Basic idea: metasymbols in the pattern describe sets of strings that match that portion of the pattern
  - metalinguage: a language for describing languages
  - metasymbol: a symbol of the metalinguage
  - Formally, symbols / metasymbols ∈ Ø

Regular Expressions (cont’d)

- Example (1): period is a metasymbol that matches any language symbol
  - "RNA" matches any 4-character string that ends with "RNA"
- Example (2): symbols enclosed in brackets define a set
  - "[^tRNA]" matches only "tRNA" or "rRNA"
Quoting Metasymbols

- In implementations of regular expressions, ordinary characters are used as metasymbols.
- To distinguish between symbols and metasymbols, "quote" the metasymbol if it is part of the pattern.

```bash
$ grep "No." NC_000913.gbk
Prints lines containing "No.", "Not classified", ...
```

Perl RE Elements

- An expression can have a modifier to specify how many times a substring can be matched:
  - `+` one or more
  - `*` zero or more
  - `{n}` exactly n
  - `{n,m}` between n and m

Examples:
```
/AB+C/  AC, ABC, ABBBC, ...
/{ACGT}(3)/ a codon
```

Perl RE Elements (cont'd)

- Perl has several predefined character classes:
  - `\d` digit
  - `\w` word character (letter or digit)
  - `\s` space, tab, newline, etc

- There are abbreviations for defining sets:
  - `{a-z}` lower case letters

- Anchors specify positions for defining sets:
  - `^` at the beginning
  - `\$` string begins and ends with a
Extracting Substrings

- Perl allows you to capture segments of the text that match designated portions of the pattern
- Strings that match pattern elements enclosed in parentheses are returned in an array

```perl
$s =~ /\w+\s+=\s+\w+/;
```  

true if $s contains "name = value"

```perl
($x,$v) = ($s =~ /\((\w+)\s+=\s+(\w+)/);
```  

assigns name part to $x, value to $v

Lazy Match

- One last point about Perl regular expressions:
  - The matching algorithm is "greedy"
  - It uses as much of the input string as it can to match with any pattern element
  - Use ? to tell Perl to use as little of the string as possible

```perl
($x,$y) = ($s =~ /(.+);(.+)/);
```  

divides line at last semicolon

```perl
($x,$y) = ($s =~ /(.+?);(.+)/);
```  

divides line at first semicolon

Limitations of Regular Expressions

- Regular expressions have limited "power" to describe patterns or find things in context
- Example: a regular expression cannot describe palindromes
  - palindromes are strings of the form ST or S.T where
    - T = reverse(S)
    - intuitive explanation: we need a stack or memory to hold S

Language

- In formal language theory, a sentence is a string of symbols from a finite alphabet
- A language is a set of sentences
- Example: strings of nucleotides
  - SŒ{A,C,T,G}+
  - The sentences of the language are defined by a grammar, a set of rules that specify how to construct the sentences

Grammar Rules

- Symbols used in grammar rules are either
  - nonterminal, i.e. metalanguage symbols used as names of syntax classes, or
  - terminal, i.e. symbols of the language

- Some conventions for writing nonterminals:
  - upper case letters
  - italics
  - names in <brackets>

Grammar Rules (cont'd)

- A rule specifies how a nonterminal can be replaced by an equivalent string
- Example:
  - A {} xyz
  - Means "(nonterminal) A can be replaced by the sequence of terminals xyz"
  - The language defined by a grammar is the set of strings that can be derived from a designated nonterminal through a series of replacements
Example: Expression Grammar #1

- Here is a grammar that defines simple arithmetic expressions:

\[
E \rightarrow T \\
T \rightarrow F \\
T \rightarrow T \cdot F \\
E \rightarrow E + T \\
T \rightarrow 0 \\
T \rightarrow 1 \\
F \rightarrow (E) \\
F \rightarrow D \\
D \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
\]

E, T, and F are nonterminals. +, *, (, ), and digits are terminals.

Expression Example

S \rightarrow T means “apply a grammar rule to rewrite S as T”

\[
E \rightarrow T \\
T \rightarrow F \\
T \rightarrow T \cdot F \\
E \rightarrow E + T \\
T \rightarrow 0 \\
T \rightarrow 1 \\
F \rightarrow (E) \\
F \rightarrow D \\
D \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 \\
\]

In each step the nonterminal of the rule used is shown in red.

More Metasymbols

- Some common symbols used when writing grammar rules:
  - | alternatives
  - [] optional form
  - ... zero or more

- The expression grammar written with these symbols:

\[
E \rightarrow (E) \\
T \rightarrow (T) \\
F \rightarrow (F) \\
D \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
\]

Parsers

- The previous example showed how to generate a sentence from a grammar
- Grammars also define the inverse: how to recognize sentences defined by a grammar

A parser is a program that analyzes a string according to a grammar

Parsers (cont’d)

- A "pure" parser just answers the question, "does string S belong to the language defined by grammar G?"
- Other applications augment the grammar rules with actions to be taken when a rule is applied:
  - compilers
  - command line processors
- Any application that processes complex text can be structured as a parser
  - formalize the document structure as a grammar
  - use parsing techniques to organize the text processing

Aside: Formal Language Theory

- Languages can be divided into classes according to the complexity of grammar rules

U = “unrestricted”

No limitations on the structure of rules

Chomsky hierarchy
Aside: Formal Language Theory

Languages can be divided into classes according to the complexity of grammar rules.

$S = \text{“context-sensitive”}$

Rules of the form $\alpha \beta \gamma$, where $\alpha$ and $\beta$ are any strings and $\gamma$ is not.

Chomsky hierarchy

$C = \text{“context-free”}$

Rules of the form $\alpha A \beta$, where $A$ is a single nonterminal on the LHS.

Chomsky hierarchy

Aside: Formal Language Theory

Languages can be divided into classes according to the complexity of grammar rules.

$R = \text{“right linear”}$

Rules of the form $A \rightarrow \alpha \beta$, where $\alpha$ and $\beta$ are any strings.

Chomsky hierarchy

Regular Expressions Again

Using methods of formal language theory:

- Languages defined by regular expressions* are equivalent to right linear languages.
- Palindromes cannot be parsed by right linear (or equivalent) grammars.
- Languages defined by context-free grammars require a parser that uses a stack.
- Perl REs are augmented with constructs that make them slightly more powerful than formal REs, but not as powerful as CFGs.

Perl REs are augmented with constructs that make them slightly more powerful than formal REs, but not as powerful as CFGs.

CFGs in Bioinformatics

- A stochastic context-free grammar (SCFG) is a formal system in which rules are assigned probabilities.
- A parse is probabilistic -- rules applied at each step are chosen based on their probabilities.
- SCFGs have been used to analyze DNA strings, e.g. to search for tRNA genes.
- We will (hopefully) look at this application later in the term.

CFG Parsers

- There are several methods for writing parsers for languages defined by CFGs.
- “Compiler compilers” are software tools for writing parsers.
  - Input = grammar
  - Output = C program to parse strings according to the grammar.
- Unix tools:
  - yacc
  - bison (GNU)
CFG Parsers (cont’d)

For smaller projects, an ad hoc parser works well
- Write one function per grammar rule:
  - each function reads from the input stream
  - a function "consumes" the portion of the stream that corresponds to its rule
  - build in "actions" to the body of the function
  - actions are triggered as parts of the RHS are evaluated

Example -- a "feature" could be structured as a name, location, and sequence of one or more attributes:
- Feature :: string Loc AttrList
- Write a function named parseFeature() that
  - reads a string (the feature type)
  - calls parseLoc() to read a location (e.g. two integers separated by ".")
  - calls parseAttrList() to read the sequence of attributes
- Actions:
  - Save name, location, etc on return from those rules

Pitfalls

- Potential trap: the parser might get caught in an infinite loop, or require "backtracking" to get out of trouble
- Expression grammar example:
  - $E \rightarrow E + T$
  - $E \rightarrow T$
  - Which rule should the parser choose? If it makes the wrong decision, it will have to "back up" and try again
  - Following the instructions on the previous slide, the first thing parseE() will do is call parseE()......

Grammars for Top-Down Parsing

- There are guidelines for (re-)organizing grammar rules so they are suitable for the recursive outline described earlier
  - For each nonterminal that has a choice, make sure the next input is sufficient to make the right choice
  - Always consume input characters before making a recursive call
  - For our feature parse: the grammar has been defined for you...

Project #2: Feature Parser

- The next class project will be a C++ program that parses the feature section of a Genbank report
- Goal:
  - produce a list of feature types
  - for each feature, list the attributes
  - use the output to understand the contents of the file
- Usage:
  - % fp < NC_000913.gbk

Project #2 (cont’d)

- The program will skip over the header section
- It will use a grammar defined for top-down recursive parsing to find the attributes of each feature
  - features will be represented by objects
  - parser will insert objects into a list
- It will return the sequence (last part of the file) as a single C++ string
Project #2 (cont’d)

- The application that calls the parser
  - defines the list passed to the parser
  - scans the list after parse to print features
  - (Optional) uses feature locations to extract sequences, write a Fasta format sequence file
- Use the parser to answer questions about the *E. coli* genome -- number of genes, types of genes, etc

Program Structure

- The project tar file will have
  - outline of fp.C, the main program
  - definition and implementation of Feature class
  - outline of GBParser, an object that implements the parser logic
  - other classes (e.g. GeneticCode) for extra credit projects
- Your job: fill in missing pieces, compile and test the program

Aside: Why Objects?

- Object-oriented programming has two benefits for this project:
  - defining features as objects allows for definition of class hierarchy, i.e. base class has attributes of all features, derived class specialize.
  - EMBL reports are similar to Genbank reports; a class named EMBLParser, derived from GBParser, can reuse a lot of existing code.

Extra Credit

- Extra Project #1: Output a Fasta file of protein sequences
  - Use the “translation” attribute of gene features
  - Include species name, sequence ID, location information in the define
- Extra Project #2: Fasta file of nucleotide sequences
  - Use location information to extract substring of DNA sequence returned by parser
  - Deal with introns, complementary strands, …