Languages, Grammars and Parsing

Robert Keller
September 2012

What are These?

- **Language**: A set of strings.
- **Grammar**: A formal way to define a language.
- **Parser**: A program that determines whether or not a given string is in the language.

More Detail on Languages

- A language is a **set of strings** of symbols from a finite set called the **alphabet**.
- When we show languages, we usually don’t show the strings with quotes, as a convenience.

Example 1

The language of all U.S. zipcodes

{00501, ..., 91711, ..., 99950}

alphabet = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

This language is finite.

Example 10

The language of all binary numerals without unnecessary leading zeros

{0, 1, 10, 11, 100, 101, 110, 111, ...}

alphabet = {0, 1}

This language is infinite.

Example 11

- The language of all “2-adic” numerals

{ε, 1, 2, 11, 12, 21, 22, 111, 112, 121, ...}

ε stands for the **empty string**

alphabet = {1, 2}

This language is infinite.
Example Scrabble

The language of official Tournament Scrabble words =
{aa, aah, aahed, ..., zyzzyvas, zzz}

alphabet = {a, b, c, ..., z}
This language is finite.

Example WBP

• WBP = Well-Balanced Parenthesis Strings
  
  • { (), (()), ((())), (())(), (()())(), ... }

  • alphabet = { ( , ) }

  • This language is

Grammars

• A grammar is a formal (i.e. mechanical) way of defining a language.

• It is a special kind of inductive definition, something we’ve seen before (think S expressions).

• It can also be viewed as a state-transition system, also déjà vu (think Picobot).

• It is also a form of rewriting system, analogous to evaluating a function.

A Grammar Has 4 Parts

• An alphabet Σ, called the terminal alphabet.

• An alphabet N, called the non-terminal alphabet. N does not overlap with Σ.

• A finite set of rules (also called “productions”).

• A start symbol, always a member of N.

Rules

• A rule indicates how a non-terminal symbol within a string can be “rewritten”, i.e. be replaced by a string.

• Only non-terminal symbols can be rewritten.

• Suppose there is a rule:
  
  \[ \text{S} \rightarrow ( \text{L} ) \]

  This is read “S rewrites as ( L )”, or “S produces ( L )”.

Non-Determinism

• We have choices.

• For a given symbol, we may have more than one rule with that symbol as the left-hand side, e.g.

  \[ \text{L} \rightarrow \text{A} \text{L} \]
  
  \[ \text{L} \rightarrow \epsilon \]

  where ε is the empty string

• Grammars are easy if you follow the rules.
Grammar as State-Transitions

• A grammar defines a state-transition system.

• The states are strings from the combined alphabet $\Sigma \cup N$.

• The transitions between states are defined on the next page.

• The initial state is the start symbol, say $S$.

Transitions

One state, a string $\ldots X \ldots$ containing a non-terminal $X$, can make a transition to another state, $\ldots \alpha \ldots$ where $\alpha$ is a string of symbols in $\Sigma \cup N$ exactly when there is a rule $X \rightarrow \alpha$ (rule)

When this is possible, we write $\ldots X \ldots \Rightarrow \ldots \alpha \ldots$ (transition)

Example

• Suppose the rules are
  $S \rightarrow (L)$
  $L \rightarrow SL$
  $L \rightarrow \varepsilon$

• Then here are some possible transitions
  $(L) \Rightarrow (SL)$ using rule $L \rightarrow SL$
  $(L) \Rightarrow (\varepsilon)$ using rule $L \rightarrow \varepsilon$
  $(SL) \Rightarrow (SSL)$ using rule $L \rightarrow SL$
  $(SL) \Rightarrow (S)$ using rule $L \rightarrow \varepsilon$
  $(SL) \Rightarrow ((L)L)$ using rule $S \rightarrow (L)$

The Yield of a Grammar

• The yield of a grammar is the set of all states reachable from the start symbol by applying 0 or more rules in sequence.

  • This can be done systematically by considering each candidate rule in turn for each possible left-hand side symbol in a state, beginning with the string consisting of just the start symbol.

Yield Example

• Consider the alphabets used previously, start symbol $S$, and rules

  • The first part of the yield is

<table>
<thead>
<tr>
<th>step</th>
<th>string</th>
<th>from state</th>
<th>using rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.</td>
<td>S</td>
<td>start symbol</td>
<td>N/A</td>
</tr>
<tr>
<td>1.</td>
<td>(L)</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2.</td>
<td>(L)</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3.</td>
<td>(SL)</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>4.</td>
<td>(S)</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Quiz

• Add the next 4 lines to the yield table.

<table>
<thead>
<tr>
<th>step</th>
<th>string</th>
<th>from state</th>
<th>using rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.</td>
<td>(SL)</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>4.</td>
<td>(S)</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>5.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Grammar to Language

- Given a grammar, the language generated by the grammar is subset of the yield containing only terminal symbols. We can construct the language by constructing the yield, collecting only the terminal strings as we go.
- The language generated by the previous grammar contains:
  - ()
  - ()()
  - (())()
  - (())(())
  - (())(())
  - etc.

Parsing

- Parsing addresses the problem:
  Based on a grammar G:
  Given a string \( x \), is \( x \) in the language generated by G, or not?
  \( x \in L(G) \)?
- There is an obvious algorithm for parsing (what?) but it is generally too slow for practical use.

Determinism in Parsing

- Although we described grammars as forming a generally non-deterministic system, we want parsing to be as deterministic as possible.
- Ideally the time taken is \( O(n) \) where \( n \) is the length of the string \( x \).

Parsing by Recursive Descent

- If the grammar is constructed the right way, there is an easy way to parse its language, using recursion.
- Illustrate with the previous grammar:

  1. \( S \rightarrow ( L ) \)
  2. \( L \rightarrow \epsilon \)
  3. \( L \rightarrow S \ L \)

Parsing an Input String

- Given an input string, such as 
  
  \( ( ) ( ) ( ) ( ) \)

we want to know “is this string in the language?”

In other words, is there a series of states of the form

\[ S \Rightarrow \ldots \Rightarrow \ldots \Rightarrow ( ) ( ) ( ) \]

[abbreviated \( S \Rightarrow^* ( ) ( ) ( ) ) \)]

To answer such questions, use the grammar to construct a set of parse functions.

Parse Functions

- There will be one parse function parse-\( X \) for each non-terminal \( X \).
- For this grammar, we will have two functions:
  
  parse-S
  
  parse-L

  1. \( S \rightarrow ( L ) \)
  2. \( L \rightarrow \epsilon \)
  3. \( L \rightarrow S \ L \)
Parse Function Jobs

- On each call, a parse function has a string x as input.
- The string x will be a suffix of the original input that has not yet been parsed. (The original input is a suffix of itself.)
- The job of the parse function is to determine whether a prefix of x can be generated by the corresponding non-terminal.

Parsing Function Picture

Parser Function Hunger

- There may be more than one y that works. Our convention is that the parse function will always prefer the longest such y.

Parse Function Jobs

- Here x is any string.
- (parse-S x) determines whether a prefix of x can be generated from S.
- (parse-L x) determines whether a prefix of x can be generated from L.
- These functions will be mutually recursive.

Parse-S

Anything S generates must start with the only production with S on the left:

\[ S \rightarrow (L) \]

If S ultimately generates a prefix of string x, then x must start with \( ( \). So (parse-S x) will check that first.

Note that x could be empty, in which case (parse-S x) fails.

But if x begins with \( ( \), say x = \( (y \) for some y, then parse-S calls (parse-L y) ...

parse-S calls parse-L

- parse-S seeing input \( (y \) calls (parse-L y).
- (parse-L y) could do one of the following:
  - fail: y can't be generated from L
  - succeed: Some prefix u of y can be generated from L, with residue v, i.e. \( y = uv \) and u is generated by L.

If parse-L succeeds, it is back to parse-S, to determine whether z begins with a matching \( ( \).

If so, (parse-S x) succeeds, leaving the residue for its caller.

If there is no matching \( ), (parse-S x) fails with residue x.
parse-L

• parse-L has to worry about two rules:
  \[ L \rightarrow S \ L \] and \[ L \rightarrow \varepsilon. \]

• Remember that parse functions are "hungry":
  they want to "eat" as much input as possible.

• So parse-L begins by immediately calling
  parse-S.
  If that succeeds, then parse-L calls itself recursively.
  If not, parse-L succeeds without eating any input,
  corresponding to the second rule.
  Thus parse-L always succeeds, one way or the other,
  whereas parse-S may fail.

Implementation Details

Each parse function must return two things:
1. An indication of whether it succeeded or
   failed.
2. The residual unparsable input.

As we plan to code these functions using racket, we’ll create a data abstraction
called Outcome.

Outcome data abstraction

• For now, an Outcome consists of only two
  things:
    success or failure
    residue of input

In future applications, we could add more things
to an Outcome, which is one of the reasons we
want a data abstraction and not a plain old list.

Constructors for an Outcome

(define (make-outcome result residue)
  (list result residue))

(define (succeed residue)     for general use
  (make-outcome 'success residue))

(define (fail residue)        for general use
  (make-outcome 'failure residue))

Accessors for an Outcome

(define (get-result Outcome) internal
  (first Outcome))

(define (get-residue Outcome)
  (second Outcome))

(define (failed? Outcome)     for general use
  (equal? 'failure (get-result Outcome)))

(define (succeeded? Outcome)
  (equal? 'success (get-result Outcome)))

Two other housekeeping functions

(define (left-paren? char) (char=? #\( char))

(define (right-paren? char) (char=? #\) char))

These prevent the code from being cluttered with
bare "magic" characters.
; Parse function for rule S -> ( L )
(define (parse-S input)
  (cond
    [null? input] (fail input))
  [(left-paren? (first input))
    (let (
      (L1 (parse-L (rest input))) ;; can't fail
     )
      (cond
        [null? (get-residue L1)] (fail input))
        [(right-paren? (first (get-residue L1)))
         (succeed (rest (get-residue L1)))]
        [else (fail input)])
    )
  ]
  [else (fail input)]))

; Parse function for rules L -> S L and L -> empty
(define (parse-L input)
  (let (
    (S1 (parse-S input))
    )
    (if (succeeded? S1)
      (let (
        (L2 (parse-L (get-residue S1))) ;; can't fail
        )
        (succeed (get-residue L2))
      )
      (succeed input)))
  )

; Parse function for rules L -> S L and L -> empty
(define (parse input-string)
  (let* (
    (outcome (parse-S (string->list input-string)))
    (residue (get-residue outcome))
    )
    (cond
      [(and (succeeded? outcome) (null? residue))
       "fully successful"
      ]
      [(succeeded? outcome)
       (string-append "successful, with residue: "
       (list->string residue))
      ]
      [else "unsuccessful"]))

; Some Unit Tests
(check-expect (parse "()")            "fully successful")
(check-expect (parse "(()())")        "fully successful")
(check-expect (parse "(()()())")      "fully successful")
(check-expect (parse "((())())")      "fully successful")
(check-expect (parse "((())(()))")    "fully successful")
(check-expect (parse "((()())(()))")  "fully successful")
(check-expect (parse "(()()))")       "successful, with residue: )")
(check-expect (parse "()()")          "successful, with residue: ()")
(check-expect (parse "(()())")        "successful, with residue: ()")
(check-expect (parse "(")             "unsuccessful")
(check-expect (parse ")")            "unsuccessful")
(check-expect (parse ")("           "unsuccessful")