Languages, Grammars, Parsing

Robert Keller
September 2010

Definitions

- **Alphabet:** A set of characters, such as

  {a, …, z, A, …, Z, 0, … 9, _, $}

- **String:** A finite ordered sequence of 0 or more characters.
- **Language**: Any set of strings over an alphabet. (The set could be infinite.)
- **Grammar**: A finite means of presenting a language.
- **Parser**: A program that determines whether a string is a member of a specific language. The operation of the program is called “parsing”.

Language Examples

- A finite language: The set of all valid zip-codes.

- An infinite language: The set of all valid Racket programs.

Grammars

- Grammars are need to precisely specify infinite languages.
- Unlike finite languages, we cannot simply list every string in the language.
- Grammars can also be used to specify finite languages more succinctly.

Grammar Components

- A grammar has 4 components, (Σ, N, S, P):
  - An alphabet, Σ, called the **terminal alphabet**. Strings in the language are strings over the terminal alphabet.
  - Another alphabet, **N**, called the **non-terminal alphabet.**
  - The **start symbol**, **S**, always a member of N.
  - A set of **productions**, **P**, as defined next.

Productions

- A production is a rule stating how a non-terminal can be replaced with a string. The non-terminal being replaced is on the LHS of an $\rightarrow$; the replacing string is on the RHS.
- The possible replacements are not necessarily unique.
- Examples of two productions:
  - S $\rightarrow$ '(' S ')'   Here '(' and ')' are non-terminals.
  - S $\rightarrow$ ε Here ε stands for the **empty string**.

Examples of Replacement

- Using the preceding two productions:
  - S → '(' S ')'
  - S → ε
- some replacement sequences (where ⇒ indicates a replacement occurs) are:
  - $\underline{S} \Rightarrow \varepsilon$ [remember ε is the empty string]
  - $\underline{S} \Rightarrow (\underline{S}) \Rightarrow ()$ [ε has no characters]
  - $\underline{S} \Rightarrow (\underline{S}) \Rightarrow ((\underline{S})) \Rightarrow (())$
- The underscores show which non-terminals were rewritten.
- The entire sequence of rewrites is called a **derivation**.

| Abbreviation

- A vertical bar | (read “or”) can be used to abbreviate several productions with the same LHS.

- Example: The two productions
  - S → ‘(‘ S ‘)’
  - S → ε

  can be abbreviated:
  - S → ‘(‘ S ‘)’ | ε

Grammars are “Non-Deterministic”

- Grammars indicate which replacements are allowed.
- They don’t necessarily specify a unique-replacement, although they can.
- A grammar can be used as a “blueprint” for a parser program.

The Language Generated by a Grammar

- Every grammar G generates a unique language (set of strings) L(G).
- The language generated is a subset of a larger set of strings, called the **yield** of the grammar.

Yield of a Grammar

- The **yield** Y(G) of a grammar G is a set of strings defined by induction.
  - The string consisting of the start symbol S is in Y(G).
  - If $xAz \in Y(G)$, and the grammar has a production $A \rightarrow y$, then $xyz \in Y(G)$.
  - The only strings in Y(G) are those obtainable by a finite sequence of applications of the above rules.
- Viewed another way, the yield is the set of strings for which there is **at least one derivation**.

The Language of a Grammar

- The language L(G) of a grammar is the subset Y(G) consisting of only **terminal** symbols.

Example

- Terminal alphabet: {‘(‘, ‘)’}
- Non terminal alphabet: {S}.
- Start symbol: S
- Productions
  - S → ‘(‘ S ‘)’ | ε
- Yield: {ε, (S), (), ((S)), (()), (((S))), ((())), …}
- Language: {ε, (), (()), ((())), …}

Example

- Terminal alphabet: {'(', ')'}
- Non terminal alphabet: {S}.
- Start symbol: S
- Productions
  - $S \rightarrow$ '(' S ')' | SS | ε
- Yield:
- Language:

Grammar for S-Expressions

- Terminal alphabet: {'(', ')'} $\cup$ D
  where D is a set of symbols disjoint from {'(', ')'}.
- Non terminal alphabet: {S, C, A, L}.
- Start symbol: S
- Productions
  - $C \rightarrow \sigma$ for each σ in D
  - $A \rightarrow CA \mid C$
  - $S \rightarrow A \mid$ '(' L ')'
  - $L \rightarrow SL \mid \varepsilon$

Example S-Expression Derivation

- Suppose D = {'a', 'b', 'c', '1', '2'}
- Productions
  - C → a | b | c | 1 | 2
  - A → CA | C
  - S → A | '(' L ')'
  - L → SL | ε
- A derivation:

$$\underline{S} \Rightarrow (\underline{L}) \Rightarrow (\underline{S}L) \Rightarrow ((\underline{L})L) \Rightarrow (()\underline{L}) \Rightarrow (()\underline{S}L)$$

$$\Rightarrow (()\underline{A}L) \Rightarrow (()\underline{C}AL) \Rightarrow (()b\underline{A}L) \Rightarrow (()b\underline{C}L)$$

$$\Rightarrow (()b1\underline{L}) \Rightarrow (()b1)$$

Syntactic Categories

- Each non-terminal can be viewed as defining a set of **syntactic categories**, which are themselves languages.
- For the S-expression example:
  - $C \rightarrow \sigma$ C defines the set of single **characters.**
  - $A \rightarrow CA \mid C$ A defines the set of **atoms.**
  - $S \rightarrow$ '(' L ')' $\mid A$ S defines the set of **S-expressions.**
  - $L \rightarrow SL \mid \varepsilon$ L defines the set of **list contents**.

Parsing

- A parser is a program that determines, for any input string over the alphabet, whether or not the string is in the language.

- Example (for the S-expression language):
  - (()b1) parser says "yes"
  - ()b1) parser says "no"

Naïve Parsing

- Given an input string, systematically generate the yield of a grammar, roughly in order of **increasing length**.
- If the input string is generated, say "yes".
- If the strings generated have gotten "long enough", so that the input string cannot possibly be generated, say "no".

Recursive Descent Parsing

- Naïve parsing is generally too inefficient to be practically useful.
- Recursive-descent parsing is a more efficient method.
- The grammar must be designed with recursive-descent parsing in mind.

Recursive Descent Example

- Consider S expressions once more:
  - $C \rightarrow \sigma$ C defines the set of single **characters.**
  - $A \rightarrow CA \mid C$ A defines the set of **atoms.**
  - $S \rightarrow$ '(' L ')' $\mid A$ S defines the set of **S-expressions.**
  - $L \rightarrow SL \mid \varepsilon$ L defines the set of **list contents**.

- The **top-level goal** is to determine whether the input is an S expression.

Parse Functions

- For an appropriately-designed grammar, we can parse by associating a parse function with each non-terminals productions.
- These will generally rely heavily on **mutual recursion**.
- Each parse function has as its argument the **remaining unparsed input** (RUI), which it will parse left-to-right.
- At the **top-level**, the RUI is the entire input.

Parse Functions for S Expressions

- Productions for the start symbol, S.
  - S → ‘(‘ L ‘)’ | A S defines the set of **S-expressions.**
- Corresponding parse function, **parse-S**:
  - If the RUI is empty, fail.
  - **If** the next symbol in the RUI is ‘(‘, then:
    - Get the result of parsing an L (new RUI is returned).
    - Check that there is a next symbol and it is ‘)’. If so, succeed. Otherwise fail.
  - **Otherwise** the result is the result of parsing an A.

Parse Functions for S Expressions

- Productions for the atom symbol, A.
  - A → CA | C   A defines the set of **atoms.**
- Corresponding parse function, **parse-A**:
  - Call parse-C, to get a character from the RUI. If that fails, then fail.
  - If that call succeeded, then call A on the new RUI. If that succeeds, return success. If it failed, then return success anyway, but keep the RUI from the most recent successful call.

Parse Functions for S Expressions

- Productions for the character symbol, C.
  - C $\rightarrow \sigma$ where σ is a non-parenthesis character
- Corresponding parse function, **parse-C**:
  - If the RUI is empty, fail.
  - If the RUI begins with a non-paren character, succeed.

Parse Functions for S Expressions

- Productions for the character symbol, C.
  - L → SL | ε   L defines the set of **list contents**
- Corresponding parse function, **parse-L**:
  - If the RUI is empty, succeed.
  - Call parse-S. If that fails, succeed keeping the original RUI.
  - If the call succeeded, return the result of calling parse-L recursively.

Structure of Parse Functions

- A parse function has one argument, the RUI (remaining unparsed input).
- The parse function returns 2 things:
  - an indication of success or failure
  - the new value of the RUI.

Data Abstraction for Return Values

```
(define success 'success)
(define failure 'failure)

; use rule for instrumentation purposes

(define (succeed rule newRUI)
       (list success rule newRUI))

(define (fail rule RUI)
   (list failure rule RUI))

(define (success? result)
   (equal? success (first result)))

 (define (failure? result)
   (equal? failure (first result)))

(define (residualUI result)   (third result))
```

Parse Function Implementation

```
; Productions for "Character":  C -> a|b|c|...|z

(define (parse-C RUI)
   (cond     ((null? RUI) (fail 'C RUI))                       ; no more input
     ((non-paren? (first RUI)) (succeed 'C (rest RUI)))         ; one of the desired chars
     (else (fail 'C RUI))))                                     ; anything else

; Productions for "Atom": A -> CA | C
(define (parse-A RUI)
   (if (null? RUI)
           (fail 'A RUI)                                         ; no more input
           (let (
               (C-result (parse-C RUI))                          ; try a char
                )
        (if (success? C-result)                                  ; have a char
           (let (
               (A-result (parse-A (residual  C-result)))         ; recurse
               )
             (if (success? A-result)
                 (succeed 'A (residual A-result))                ; case CA
                 (succeed 'A (residual C-result))))              ; case C
           (fail 'A RUI)))))                                     ; anything else
```

Parse Function Implementation

```
;  Productions for "List Content": L -> SL | empty

(define (parse-L RUI)
   (if (null? RUI)
        (succeed 'L RUI)                          ; no more input, empty case
       (let (
            (S-result (parse-S RUI))          ; try an S
            )
        (if (success? S-result)
             (parse-L (residual S-result))   ; case SL
             (succeed 'L RUI)))))              ; empty case
```

Parse Function Implementation

```
;  Productions for "S Expression": S -> (L) | A

(define (parse-S RUI)
   (cond
     ((null? RUI) (fail 'S RUI))                         ; no more input, fail
    ((left-paren? (first RUI))
     (let* (
          (L-result (parse-L (rest RUI)))                ; have (, try L
           (residue (residual L-result))
           )
       (if (success? L-result)
           (if (and (not (null? residue))
                       (right-paren? (first residue)))   ; have (L, try ')’
               (succeed 'S (rest residue))               ; case (L)
               (fail 'S RUI))                            ; (L, but no ')', fail
           (fail 'S RUI))))                              ; ( but no L, fail
    (else
       (let (
            (A-result (parse-A RUI))                     ; no (, try A
            )
        (if (success? A-result)
            (succeed 'S (residual A-result))             ; have A, success
            (fail 'S RUI))))))                           ; no A, fail
```

What about Whitespace?

- Although it is usually left to informal treatment, a grammar might not be totally accurate unless whitespace is taken into account.

- For example, in our S-expression grammar so far, we couldn't parse a string of the form (ab cd), because the grammar will not recognize the whitespace gap.

Two Kinds of Whitespace

- There are places where whitespace is **essential** and places where it is **optional**.
- When one whitespace character is essential, additional whitespace characters are often optional.

Delimiters

- A delimiter is something that breaks up the flow of text.
- Whitespace can be a delimiter, but so can other characters such as ‘(’ and ‘)’ in the case of S-expressions.
- To include all possibilities would significantly complicate the grammar.

Whitespace Productions

- Let W stand for optional whitespace. The productions are:

  W $\rightarrow \varepsilon$ | ‘ ’ W (i.e. ‘ ’ is a **blank** character)

- Other characters, such as **tab** and **formfeed** might also be considered whitespace.

- Let E stand for **essential** whitespace. We’ll assume that essential is **generally followed by optional**.
  The production for E then would have the form
  E $\rightarrow$ ‘ ’ W
  where W is as defined above.

S-Expressions Revised

- Original grammar:
  - $C \rightarrow \sigma$ C defines the set of single **characters.**
  - $A \rightarrow CA \mid C$ A defines the set of **atoms.**
  - $S \rightarrow$ ‘(‘ L ‘)’ | A S defines the set of **S-expressions.**
  - $L \rightarrow SL \mid \varepsilon$ L defines the set of **list contents**.
- Modified grammar,
  with **W as optional whitespace**:
  - $C \rightarrow \sigma$
  - $A \rightarrow CA \mid C$
  - $S \rightarrow$ W ‘(‘ L W ‘)’ W | WAW
  - $L \rightarrow SL \mid W$
- We don’t need to add W to the L productions, because it is accounted for in the S productions.

Optional-Whitespace Parser

```
; optional-whitespace parser

(define (skip-white RUI)
      (cond
        ((null? RUI) '())
        ((char-whitespace? (first RUI) (skip-white (rest RUI)))
        (else RUI)))

; char-whitespace? is built-in
```

Revised S-Expression Parse Function

```
; Productions for "S Expression": S -> W ( L W ) W | W A W  where W represents optional
     whitespace
(define (parse-S RUI)
   (let ((RUI (skip-white RUI)))
   (cond
     ((null? RUI) (fail 'S RUI))                              ; no more input, fail
     ((left-paren? (first RUI))
      (let* (
           (L-result (parse-L (skip-white (rest RUI))))       ; have (, try L
           (residue (skip-white (residual L-result)))
           )
        (if (success? L-result)
           (if (and (not (null? residue))
                    (right-paren? (first residue)))           ; have (L, try ')'
              (succeed 'S (skip-white (rest residue)))        ; case (L)
              (fail 'S RUI))                                  ; (L, but no ')', fail
           (fail 'S RUI))))                                   ; ( but no L, fail
     (else
        (let (
            (A-result (parse-A RUI))                          ; no (, try A
            )
         (if (success? A-result)
            (succeed 'S (skip-white (residual A-result)))   ; have A, success
            (fail 'S RUI))))))                                ; no A, fail
```

Precedence

Semantic Considerations

- Often a parser has to do more than just determine whether a string is in the language.
- It also may be tasked with **providing a meaning** for the string.

Example: S-Expression Semantics

- The meaning of an S-expression could be the internal form used by Racket.
- The parser can construct this meaning using cons and a few other built-in functions.

Categories for Infix Arithmetic Grammars

- Define a grammar for **additive expressions**, where + is the add operator, e.g. the language is: {a, b, a+a, a+b, b+a, b+b, a+a+a, …}

- Let V be a non-terminal representing the category of “variables”.

Categories for Infix Arithmetic Grammars

- Define a grammar for **additive and multiplicative expressions**, where + and * are the add and multiply operators, e.g. the language is the previous one, union with:

  {a*a, a+a*a, a*a+a, …}

Precedence

- We want * to **have precedence** over +.
- Put another way, * has a stronger **binding strength** than +.

- a*b+c should be interpreted as if
  (a*b) + c
  not
  a * (b+c)

- This is accomplished by constructing the grammar in a particular way:
  Higher precedence operators are
  “farther from the start symbol”.

Derivation Trees

Grouping or “Associativity”

- We might want an operator such as + to be grouped a particular way:

- e.g. should a + b + c

  be interpreted as if

  (a + b) + c

  or as if

  a + (b + c)

These things can be controlled by the grammar and/or parser.