Parsing

Two Main Language Problems

- Recognition problem:
  Is a given string in the language?
- Meaning problem:
  What is the meaning of a string if it *is* in the language?

Naïve Solution to the Recognition Problem

- To determine whether string x is in the language generated by a grammar:
  - Start with the start symbol.
  - Generate strings successively by applying productions.
  - Eventually either:
    - The string x is generated, or
    - The new strings being generated all exceed x in length.
  - So we can tell whether or not x is *ever* generated.

Parsing

- Parsing seeks to solve both problems:
  - Recognition
  - Meaning
- In addition, it tries to do recognition much more efficiently than the naïve solution.

Recursive Descent Parsing

- Simplest reasonably general form of parsing.
- Works for many, but not all grammars.
- Sometimes a grammar can be transformed to enable recursive descent.
- Recall that each each auxiliary symbol in the grammar can be identified with a **syntactic category**, the set of strings that can be generated from that symbol (possibly with the help of other symbols). The meaning will derive from this idea.

Recursive Descent

- It's called "recursive" because in general grammar productions can "call" themselves or each other.
- It's called "descent" because parsing starts at the **root** of a "derivation tree" and proceeds toward the leaves.

Parse Methods

- For each auxiliary symbol in the grammar, construct a **parse method**
- Each parse method's responsibility is to recognize the longest string in the corresponding syntactic category in the remainder of the input, from the current point onward:

$$\underbrace{a\ +}_{\text{passed}}\ \underbrace{b\ *\ c}_{\text{remaining}}$$

Example

- Consider the grammar with start symbol S:
  - S → V + S | V
  - V → a | b | c
- The parse begins by trying to identify the entire input string as being in syntactic category S.
- Clearly it must find a V to start.
  - To find a V, it checks to see whether the next symbol is one of those listed.
- Having found a V, it checks to see if the next symbol is +.
  - If so, it recurses, trying to find another S.
  - If not it stops.
- After the top call to S returns, it checks to see whether there are any spurious remaining characters in the input.
  - If there are, the input is not accepted.
  - If not, the input is accepted.

Example: Success

S → V + S | V
V → a | b | c

- Suppose the input string is "a + b + c".
- Subscripts will indicate the particular instance of the method and the "argument" will indicate the unparsed remainder of the input.
- The parser calls S_1("a + b + c").
- S_1 calls V_1("a + b + c").
- V_1 identifies a, returns success and unparsed input "+ b + c".
- S_1 checks for + and finds it; therefore S_1 calls S_2("b + c").
- S_2 calls V_2("b + c").
- V_2 identifies b, returns success and unparsed input "+ c".
- S_2 checks for + and finds it; therefore S_2 calls S_3("c").
- S_3 calls V_3("c").
- V_3 identifies c, returns success and unparsed input "".
- S_3 checks for + and does not find it; therefore S_3 returns success with "".
- S_2 returns success with "".
- S_1 returns success with "". **The string is accepted.**

Example: Failure

S → V + S | V
V → a | b | c

- Suppose the input string is "a b + c".
- The parser calls S_1("a b + c").
- S_1 calls V_1("a b + c").
- V_1 identifies a, returns success and unparsed input "b + c".
- S_1 checks for + and does not find it; therefore S_1 returns success, with "b + c".
- **Since the top-level call to S_1 has returned, but there is residual input, the string is not accepted.**

A rex version of parsing

- Each syntactic category will be a rex function.
- There is one argument:
  - the unparsed input, a **list** of characters.
- There are two results:
  - success or failure indicator
    - for success: the Syntax Tree
    - for failure: FAILURE (some special value, not a syntax tree)
  - the unparsed input.

A rex version of parsing (1)

```
// parse function for auxiliary A, rules A -> V | V + A

A(input) =
    Vresult = V(input),                               // try for V

    [tree1, residue1] = Vresult,                      // separate

    failed(tree1) ? Vresult                           // V failed

  : residue1 == [] ? Vresult                          // use A -> V

  : first(residue1) == '+' ?                          // see if '+' follows

      (
       [tree2, residue2] = A(rest(residue1)),         // try A -> V + A

       failed(tree2) ? Vresult                        // use A -> V only

      : [mkTree('+', tree1, tree2), residue2]         // use A -> V + A
      )

  : Vresult;                                          // use A -> V
```

Test cases

```
test(A(explode("a")),          ['a', []]);
test(A(explode("a+b")),        [['+', 'a', 'b'], []]);
test(A(explode("a+b+c")),      [['+', 'a', ['+', 'b', 'c']], []]);
test(A(explode("a+b+c+a")),    [['+', 'a', ['+', 'b', ['+', 'c', 'a']]], []]);
test(A(explode("")),           [FAILURE, []]);
test(A(explode("+")),          [FAILURE, ['+']]);
test(A(explode("ab")),         ['a', ['b']]);
test(A(explode("a+b+")),       [['+', 'a', 'b'], ['+']]);
test(A(explode("a+b+c+")),     [['+', 'a', ['+', 'b', 'c']], ['+']]);
test(A(explode("ab+c")),       ['a', ['b', '+', 'c']]);
test(A(explode("a+b+")),       [['+', 'a', 'b'], ['+']]);
```

A rex version of parsing (2)

```
// parse function for auxiliary V, rules V -> a | b | c

V([]) => [FAILURE, []];                                   // no input

V([c | chars]) => isVar(c) ? [mkTree(c), chars];          // variable

V([c | chars]) => [FAILURE, [c | chars]];                 // not a variable


// auxiliary functions

FAILURE = "failure";
VARS = ['a', 'b', 'c'];

isVar(char) = member(char, VARS);

failed(result) = result == FAILURE;

mkTree(Var) = Var;
mkTree(Op, Tree1, Tree2) = [Op, Tree1, Tree2];

parse(string) = A(explode(string));
```

Operators + and *
with * having higher precedence

- Rules:
  - A → M + A | M
  - M → V * M | V
  - V → a | b | c
- Note that * is *analogous* to +.
  - A is to M and + as
    M is to V and *
- Therefore the *same* rule **pattern** applies to both.

rex parsing for +, * (A)

```
A(input) =
    Mresult = M(input),                                   // try for M

    [tree1, residue1] = Mresult,

    residue1 == [] ? Mresult                              // use A -> M

  : failed(tree1) ? Mresult                               // failure

  : first(residue1) == '+' ?

      ( [tree2, residue2] = A(rest(residue1)),            // try A -> M + A

        failed(tree2) ?

          Mresult                                         // use A -> M only

        : [mkTree('+', tree1, tree2), residue2]           // use A -> M + A
      )

  : Mresult;                                              // use A -> M
```

rex parsing for +, * (M)

```
M(input) =
    Vresult = V(input),                                   // try for V

    [tree1, residue1] = Vresult,

    failed(tree1) ? Vresult                               // failure

  : residue1 == [] ? Vresult                              // use M -> V

  : first(residue1) == '*' ?

      ( [tree2, residue2] = M(rest(residue1)),            // try M -> V * M

        failed(tree2) ?

          Vresult                                         // use M -> V only

        : [mkTree('*', tree1, tree2), residue2]           // use M -> V + M
      )

  : Vresult;                                              // use M -> V
```

Parsing Methods in Java

- In the Java version, we will "not need to" return the unparsed input as a value.
- We can side-effect the input stream to achieve a similar result, "using up" characters as we go.
- We can store the input stream in the parse object, rather than pass it as an argument.

Parsing Methods in Java

```
/**
 * ParseFromString is a base class for parsing from a String,
 * such as a single input line.
 */

class ParseFromString
{
ParseFromString(String input) // constructor

char nextChar()

boolean nextCharIs(char c)

char peek()

boolean skipWhitespace()
}
```

various utility methods

Additive Grammar

A V | V + A

V a|b|c|d|e|f|g|h|i|j|k|l|m
   |n|o|p|q|r|s|t|u|v|w|x|y|z

Corresponding to the grammar above, there will be two parse methods:

A()

V()

Each parses from the current point in the input.

Runnable Examples

parse/addRecursive/Additive.java

parse/add/Additive.java

parse/addMult/AddMult.java

parse/simpleCalc/SimpleCalc.java

V() method

```
/**
 * PARSE METHOD for V  a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p
 *                      |q|r|s|t|u|v|w|x|y|z
 */

Object V()
  {
  skipWhitespace();

  if( isVar(peek()) )
    {
    return makeString(nextChar());
    }
  return failure;
  }
```

makeString(char c)

```
/**
 * make a String from a char
 */

static String makeString(char c)
  {
  return (new StringBuffer(1).append(c)).toString();
  }
```

isVar()

```
/**
 * predicate defining whether its argument is a variable
 */

boolean isVar(char c)
  {
  switch( c )
    {
    case 'a': case 'b': case 'c': case 'd': case 'e': case 'f': case 'g':
    case 'h': case 'i': case 'j': case 'k': case 'l': case 'm': case 'n':
    case 'o': case 'p': case 'q': case 'r': case 's': case 't': case 'u':
    case 'v': case 'w': case 'x': case 'y': case 'z':
      return true;

    default:
      return false;
    }
  }
```

Do *not* use **arithmetic** on integer codes for this purpose.

Recursive A() method

```
/**
 * PARSE METHOD for A -> V { '+' V }
 */

Object A()
  {
  Object result;
  Object V1 = V();
  if( isFailure(V1) ) return failure;

  if( skipWhitespace() && nextCharIs('+') )
    {
    Object A2 = A();
    if( isFailure(A2) ) return failure;
    return OpenList.list("+", V1, A2);
    }
  else
    {
    return V1;
    }
  }
```

Replacing some Recursion with Iteration

"Inverse McCarthy Transformation" for Grammars with **left**-grouping

{ } is a meta-symbol meaning **"0 or more of what's inside"**

- Recursion → Iteration
- Works in some cases, not all
- Use for convenience and readability

Recursive Form

A → V | A + V
V → a | b | c

Iterative Form

A → V **{** + V **}**
V → a | b | c

both forms are "left grouping" in this example

A() method, iterative version

```
/** PARSE METHOD for A -> V { '+' V } */

Object A()
  {
  Object result;
  Object V1 = V();
  if( isFailure(V1) ) return failure;

  result = V1;

  while( skipWhitespace() && nextCharIs('+') )
    {
    Object V2 = V();
    if( isFailure(V2) ) return failure;
    result = OpenList.list("+", result, V2);
    }
  return result;
  }
```

The Additive/Multiplicative Grammar

Additive

```
A -> V { '+' V }

V -> a|b|c|d|e|f|g|h|i|j|k|l|m
       |n|o|p|q|r|s|t|u|v|w|x|y|z
```

Additive and Multiplicative

```
A -> P  { '+' P }
P -> V  { '*' V }
V -> a|b|c|d|e|f|g|h|i|j|k|l|m
       |n|o|p|q|r|s|t|u|v|w|x|y|z
```

Construct methods by analogy.

Remembering Precedence Rules

- Tighter-binding operators are introduce **further** away from the root of the grammar:

root

A → P { '+' P }

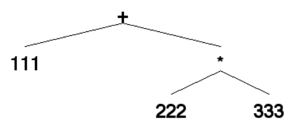
P → V { '*' V }

* binds more tightly than +

Syntax Tree Applet

Input numeric expression for syntax analysis:

111 + 222 * 333



http://www.cs.hmc.edu/courses/current/examples/java/parse/syntaxTree/SyntaxTree.html

Example: SimpleCalc

- Parses numeric expressions with +, *, ()
- Computes the *numeric* answer
- Same grammar as SyntaxTree applet

```
/**
 * SimpleCalc Parse method for A -> P { '+' P }
 */

Object A()
  {
  Object result = P();                    // get first addend
  if( isFailure(result) ) return failure;

  while( skipWhitespace() && nextCharIs('+') )
    {
    Object P2 = P();                      // get next addend
    if( isFailure(P2) ) return failure;
    try
      {
      result = Arith.add(result, P2);    // accumulate result
      }
    catch( IllegalArgumentException e )
      {
      System.err.println("error: IllegalArgumentException caught");
      }
    }
  return result;
  }
```