Review:
Languages and Grammars

- A language is a (possibly infinite) set of (finite) strings
- A grammar is a set of rewriting rules for generating languages via derivations

\[
\begin{array}{c}
S \to V | V + S \\
V \to a | b | c \\
S \to V + S \to c + S \to c + V + S \to c + b + S \to c + b + V \to c + b + c \\
S \to V + S \to V + V + S \to V + V + V \to V + V + c \to V + V + c \to c + V + c \to c + b + c \\
\end{array}
\]

Review: Derivation Trees

- The representation of a production sequence (actually, many equivalent sequences)
  - Leaves are terminals, internal nodes are nonterminals
  - Children are the items replacing the nonterminal
  - Shows the structure of the string

A Grammar for Arithmetic

- Written slightly differently (BNF), but ideas are exactly the same
- Alternatives are separated by vertical bars.
- The nonterminals are written \(<digit>\) and \(<number>\) and \(<exp>\), while the terminals include digits, +, *, and ).
- See handout for a much larger grammar, written using yet another variant of the grammar notation.
Ambiguities: Associativity

- Suppose our grammar were:
  
  \[
  S \to V \mid S+S \\
  V \to a \mid b \mid c
  \]

- What is the derivation tree for \(a+b+c\) ?

- Does it matter?

Does Grouping Matter?

- Mathematically, + is an associative operator:
  \[(a + b) + c = a + (b + c)\]

- However:
  - There are non-associative operators, such as -,
    where it does matter.
    \[(a - b) - c \neq a - (b - c)\]
  - On computers, for floating point addition,
    associativity does not always hold.

Floating Point is Not Associative

- Try this:
  
  ```
  def sumup(m, n):
      return m > n ? 0 : 1. / m + sumup(m+1, n);
  def sumdown(m, n):
      return m > n ? 0 : 1. / n + sumdown(m, n-1);
  def test(n):
      return sumup(1, n) == sumdown(1, n);
  
  map(test, range(1, 100));
  ```

- Why are there zeros here?

Ambiguities: Precedence

- Suppose our grammar were:
  
  \[
  S \to V \mid S+S \mid S*S \\
  V \to a \mid b \mid c
  \]

- What is the derivation tree for \(a*b+c\) ?

- Why are there zeros here?
Resolving Ambiguities

• One can often modify the grammar to ensure that strings have unique derivation trees.

S → V | V + S
V → a | b | c

S → P | P + S
P → V | V * P
V → a | b | c

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.
  - Keep generating strings 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.
Lexing and Parsing

- Modern compilers usually start with a lexer and a parser
  - Lexer: breaks input into tokens (words)
  - Parser: turns tokens into trees
- Today we will skip the lexer
  - Our parsers will work directly with characters.
  - The homework on the other hand...

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.
- Observation:
  - Each auxiliary symbol in the grammar can be identified with a syntactic category, the set of strings that can be generated starting from that symbol.
  - This will help give some intuition for the code.

Recursive Descent

- It’s called recursive because in general productions for a nonterminal can involve that nonterminal.
- It’s called descent because parsing starts by determining the root of the derivation tree and proceeds toward the leaves.

Parse Methods

- For each nonterminal in the grammar, construct a parse method
  - To do: recognize the longest prefix of its input in the corresponding syntactic category

    \[ a \ast b + c \]
Example

```
S → V | V + S
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 remaining characters in the input.
  - If there are, the input is not accepted.
  - If not, the input is accepted.

Example 1

```
S → V | V + S
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 2

```
S → V | V + S
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.

Recursive Descent in rex

- 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 failure: the string "failure"
    - for success: the abstract syntax tree
  - the leftover input, also a list of characters.
Parse Function for $V$ (& other stuff)

// parse function for auxiliary $V$, rules $V \rightarrow a | b | c$

$V([\]) \Rightarrow [\text{FAILURE}, [\]]; // no input$

$V([c | \text{chars}]) \Rightarrow \text{isVar}(c) ? [\text{mkTree}(c), \text{chars}]; // \text{variable}$

$V([c | \text{chars}]) \Rightarrow [\text{FAILURE}, [c | \text{chars}]]; // \text{not a variable}$

// auxiliary functions

FAILURE = "failure";

VARS = ['a', 'b', 'c'];

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

gs(result) = (result == FAILURE);

mkTree(Var) = Var;

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

parse(string) = S(explode(string));

Parse Function for $S$

// rules: $S \rightarrow V | V+S$

$S(\text{input}) \Rightarrow$

\[
\begin{align*}
(&\text{result1, residue1} = V(\text{input}), &\text{try V} \\
&\text{failed(result1) ? & \text{V failed} \\
&\text{residue1} == [\] ? & \text{S -> V used} \\
&\text{first(residue1) == '+' ? & \text{S -> V+S used} \\
&(\text{result2, residue2} = S(\text{rest(residue1)}), & \text{try S->V+S} \\
&\text{failed(result2) ? & \text{V+, but S failed} \\
&\text{result1, residue1} \rightarrow & \text{S -> V+S used} \\
&\text{mkTree('+', result1, result2), & \text{S -> V+S used} \\
&\text{residue2} \rightarrow & \text{no more options})}
\end{align*}
\]

Test Cases

parse("a+b+c") =>

\[
[\text{['+', 'a', ['+', 'b', 'c']]}], []]
\]

parse("a+b+c+a") =>

\[
[\text{['+', 'a', ['+', 'b', ['+', 'c', 'a']]}], []]
\]

parse("ab+c") => ['a', ['b', ['+', 'c']]);

parse("a+b+a") => [['+', 'a', 'b'], ['+']]

parse("+a") => [FAILURE, ['+', 'a']];

Adding Multiplication

- Note the analogy between $S$ and $P$
- Therefore the same pattern can be used to implement both
New Function $P$

```
// Rules $P \rightarrow V \mid V*P$

$P(input) =$
    [result1, residue1] = V(input),  // try V
        failed(result1) ?
    [FAILURE, residue1]  // V failed
        : residue1 == [] ?
        [result1, residue1]  // $P \rightarrow V$ used
            : first(residue1) == '*' ?
            ([result2, residue2] = P(rest(residue1)), // try $P \rightarrow V*P$
                failed(result2) ?
                [result1, residue1]  // $V*$ found, $P$ failed
                    : [mkTree('*', result1, result2),
                        residue2]  // $P \rightarrow V*P$ used
            )
        : [result1, residue1];  // $P \rightarrow V$ used
```

Revised $S$

```
// Rules $S \rightarrow P \mid P+S$

$S(input) =$
    [result1, residue1] = P(input),  // try $P$
        failed(result1) ?
    [FAILURE, residue1]  // $P$ failed
        : residue1 == [] ?
        [result1, residue1]  // $S \rightarrow P$ used
            : first(residue1) == '+' ?
            ([result2, residue2] = S(rest(residue1)), // try $S \rightarrow P+S$
                failed(result2) ?
                [result1, residue1]  // $P+$, but $S$ failed
                    : [mkTree('+', result1, result2),
                        residue2]  // $S \rightarrow P+S$ used
            )
        : [result1, residue1];  // no more options
```

Parsing Methods in Java

- Can implement recursive descent parsers in Java in exactly the same way.
  - One method for each nonterminal.
- However, in the Java version, we don't need to return the unparsed input as a value.
  - Instead, we can *side-effect* the input stream to achieve a similar result
  - Input characters are "used up" as we go.
  - Input will be implicit ("global")
- See examples/java/parse
  - e.g., `syntaxTree`

// ParseFromString is the base class for parsing from a String, such as a single input line.
```
**V() method**

```java
/**
 * PARSE METHOD for V → a|b|c|...|z
 */
Object V() {
    skipWhitespace();
    if( isVar(peek()) ) {
        return makeString(nextChar());
    }
    return failure;
}

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

**isVar()**

```java
/**
 * predicate checking whether 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;
    }
}
```

**Recursive S() method**

```java
/**
 * PARSE METHOD for S → V | V + S
 */
Object S() {
    Object V1 = V();
    if( isFailure(V1) ) return failure;
    if( skipWhiteSpace() && nextCharIs('+') ) {
        Object S2 = S();
        if( isFailure(S2) ) return failure;
        return Polylist.list('+', V1, S2);
    } else {
        return V1;
    }
}
```

"Inverse McCarthy Transformation" for Grammars with left-grouping

- In some cases, can turn recursion into iteration
  - Use for convenience and readability
- Use `{ }` as a meta-symbol meaning "0 or more of what's inside"
  - Not universal; often these are terminals

<table>
<thead>
<tr>
<th>Recursive Form</th>
<th>Iterative Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>S → V</td>
<td>V + V</td>
</tr>
<tr>
<td>V → a</td>
<td>b</td>
</tr>
</tbody>
</table>
Iterative \texttt{S()} method

```java
/** PARSE METHOD for \texttt{S -> V \{ '+V' \}} */
Object \texttt{S()} {  
  Object result;
  Object \texttt{V1} = \texttt{V()};
  if( isFailure(\texttt{V1}) ) return failure;
  result = \texttt{V1};
  while( skipWhitespace() && nextCharIs('+') ) {  
    Object \texttt{V2} = \texttt{V()};
    if( isFailure(\texttt{V2}) ) return failure;
    result = Polylist.list("*", result, \texttt{V2});
  }
  return result;
}
```

Parentheses

- Parentheses means "handle as a single unit"
- So, we can treat these as "atomic" items just like single variables

\[
\begin{align*}
S & \rightarrow P & P+S \\
P & \rightarrow V & V*P \\
V & \rightarrow a & b & c & (S)
\end{align*}
\]

SimpleCalc Example

- Parses numeric expressions with +, *, (, )
- Computes the numeric answer
- Same grammar as SyntaxTree applet
  - Only difference is the parse methods return a numeric object, instead of a representation of the syntax tree.

SimpleCalc's \texttt{S()} Method

```java
/**
* SimpleCalc Parse method for \texttt{S -> P \{ '+P' \}}
**/
Object \texttt{S()} {  
  Object result = \texttt{P()};  
  // get first addend
  if( isFailure(result) ) return failure;
  while( skipWhitespace() && nextCharIs('+') ) {  
    Object \texttt{P2} = \texttt{P()};
    // get next addend
    if( isFailure(\texttt{P2}) ) return failure;
    try {  
      result = Arith.add(result, \texttt{P2});  
      // accumulate result
    } catch( IllegalArgumentException e ) {  
      System.err.println("error: IllegalArgumentException");  
    }
  }
  return result;
}
```