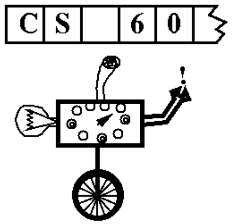


Parsing

November 2, 2001

Review: Languages and Grammars

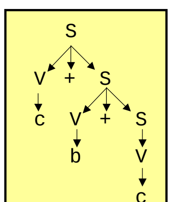
- A *language* is a (possibly infinite) set of (finite) strings
- A *grammar* is a set of rewriting rules for generating languages via *derivations*

```
S → V   |   V + S
V → a   |   b   |   c
```

S → V+S → c+S → c+V+S → c+b+S → c+b+V → c+b+c
S → V+S → V+V+S → V+V+V → V+V+c → c+V+c → c+b+c

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




```
S → V   |   V + S
V → a   |   b   |   c
```

S → V+S → c+S → c+V+S → c+b+S → c+b+V → c+b+c
S → V+S → V+V+S → V+V+V → V+V+c → c+V+c → c+b+c

A Grammar for Arithmetic

```
<digit>  ::= 0 | 1 | 2 | 3 | 4
           | 5 | 6 | 7 | 8 | 9
<number> ::=  <number><digit> | <digit>
<exp>    ::= <exp> + <exp> | <exp> * <exp>
           | ( <exp> ) | <number>
```

- 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 → V	|	S + S			
V → a	|	b	|	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 != a - (b - c)`
  - On computers, for floating point addition, associativity does not always hold.

Floating Point is Not Associative

- Try this:

```
sumup(m, n)   = m > n ? 0 : 1./m + sumup(m+1, n);
sumdown(m, n) = m > n ? 0 : 1./n + sumdown(m, n-1);
test(n) = sumup(1, n) == sumdown(1, n);

map(test, range(1, 100));

[1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1,
 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 1,
 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 0,
 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0]
```

- Why are there zeros here?

Ambiguities: Precedence

- Suppose our grammar were

S → V	|	S + S	|	S * S	
V → a	|	b	|	c	

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

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	→	V	|	S + V		
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 * 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 -> 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) = S(explode(string));
```

Parse Function for S

```
// rules:   S -> V | V+S

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

Test Cases

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

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

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

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

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

Adding Multiplication

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

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

New Function P

```
// Rules P -> V | V*P

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

Revised S

```
// Rules S -> P | P+S

S(input) =>
    [result1, residue1] = P(input),       // try P
    failed(result1) ?
        [FAILURE, residue1]               // P failed
  : residue1 == [] ?
        [result1, residue1]               // S -> P used
  : first(residue1) == '+' ?
      ([result2, residue2] = S(rest(residue1)), // try S->P+S
           failed(result2) ?
               [result1, residue1]        // P +, but S failed
         : [mkTree('+', result1, result2),
            residue2]                     // S -> 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`

Parsing Methods in Java

```
/**
 * ParseFromString is the 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)                various utility methods

char peek()

boolean skipWhitespace()
}
```

`V()` method

```
/**
 * 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()`

```
/**
 * 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

```
/**
 * 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*

Recursive Form

S → V | S + V
V → a | b | c

Iterative Form

S → V { + V }
V → a | b | c

Iterative `S()` method

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

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

  result = V1;

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

Parentheses

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

```
S → P | P+S
P → V | V*P
V → a | b | c | (S)
```

`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 `S()` Method

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

Object S() {
  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");
    }
  return result;
}
```