# CS206 Lecture 18
# Term Rewriting Code

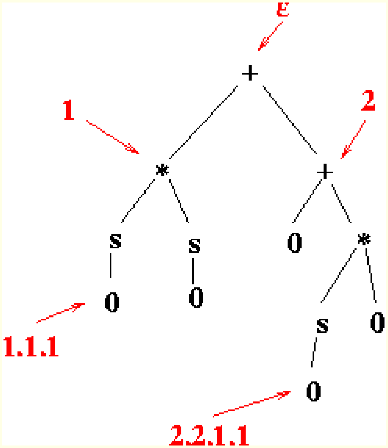
**G. Sivakumar**
Computer Science Department
IIT Bombay
siva@iitb.ac.in
http://www.cse.iitb.ac.in/~siva

Fri, Feb 14, 2003

## Plan for Lecture 18

- Normalization Strategies
- Prolog code
- Java code

# Indexing Positions/Subterms




- A position $\lambda$ identifies a subterm
- In above example $t/2.2 = s(0) * 0$
- Notation for replacing a subterm $t'$ in a term $t$ by another term $u$

$$t[\lambda \leftarrow u]$$

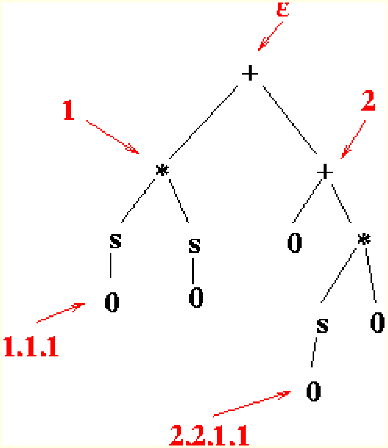
- Example:

$$t[2 \leftarrow 0] = (s(0) * s(0)) + 0$$

# Redex

Let $t$ be a term and $R$ a set of rewrite rules.

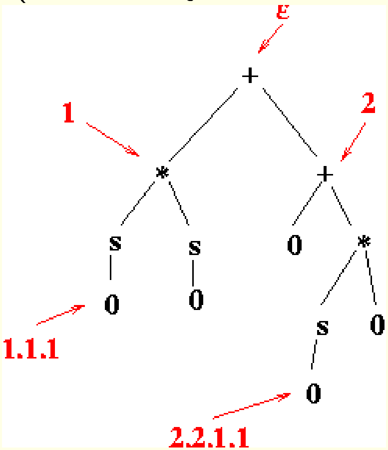
A redex in $t$ is a position $\lambda$ in $t$ where some rule of $R$ can apply. That is, $\lambda$ is a redex in $t$ if there is a rule $l \rightarrow r$ in $R$ and a substituion $\sigma$ such that $l\sigma == t/\lambda$.




has 3 redexes– 1, 2 and 2.2

# Outermost and Innermost

A redex $r$ in $t$ is **outermost** if no **prefix** of $r$ is also a redex. (Informally: no superterm can be reduced)
A redex $r$ in $t$ is **innerrmost** if there is no position $r1$ in $t$ with $r$ as **prefix** such that $r1$ is also a redex. (Informally: no subterm can be reduced)




In
2 is an outermost redex.
2.2 is an innermost redex.
1 is both innermost and outermost redex.

# Rewriting Strategy

Let $t$ be a term and $R$ a set of rules.

A reduction sequence

$$t \longrightarrow t_1 \longrightarrow t_2 \longrightarrow \ldots$$

is outermost (innermost) if each at step in the sequence a rule is applied at an outermost (innermost) redex.

A **mixed** strategy is one which is neither outermost nor innermost.

Which strategy is best?

Implement all 3 strategies.

# Term Rewriting in Java

Prolog code easier to understand?

- Separate Classes for
  - Term
  - Subst
  - Rule
- With appropriate methods

A term is an object which has methods to do things such as

- convert itself to string (for printing)
- applying a substitution on itself
- checking equality with another term
- matching with another term
- normalizing itself using some rules

# Vectors versus Arrays

Arrays are **fixed** size.

Not good when we do not know **arity** (number of arguments of a term) or we wish to input unknown number of rules etc.

Vectors are good for this.

Java has built-in Vector class which can hold a dynamic list of objects.

Useful methods.

- Vector rules = new Vector();
- rules.addElement(anyObject);
- rules.elementAt(i); //starts from 0
- int nrules = rules.size();

Home Page

Title Page

Contents

Page 8 of 18

Go Back

Full Screen

Close

Quit

# Type Casting

A vector is a list of Objects. So, when adding to a vector we can add **anything**.

When retrieving elements from a Vector we have to **typecast** it properly.

Example- let arguments of a term be stored in a vector. Then,

```
for (int i = 0;
      i < t.args.size(); i++){
 Term targ = (Term) t.args.elementAt(i);
  << do something with targ >>
  }
```

# Term.java

```
public class Term {
// the 2 important fields of any term
public String opvarname;
public Vector args;
// The Term constructor below builds
// from a String such as "f(x,0,g(y,z))"
public Term(String str){
      ... }
//methods
public boolean isvar()
public boolean isconst()
// compare with another term.
public boolean equals(Term t)
// replace one of the top level arguments in term
// to make a new term. e.g.  f(a,b).rplarg(1,b) gives f(b,b)
public Term rplarg(int j, Term narg)
// make a new copy with varnames suffixed by a number.
public Term copy(int vnum)
// apply a substitution to a term
public Term applySubst(Subst sigma)
... And many more ...
}
```

# Parsing a term from a String

No error checking below.

```
public Term(String str){
 args = new Vector(); // initialize to null Vector.
 int i1 = str.indexOf('(');
 if (i1 == -1){
  // this is a constant or a variable
  opvarname = str;}
  else {
  //  this is f(t1,..,tn) where n is arity of f
     opvarname = str.substring(0, i1);
     int paren = 1;
     while (paren > 0) {
   for (int pos = i1 + 1; pos <= str.length(); pos++){
     char ch = str.charAt(pos);
     if (ch == '(')
   paren++;
     else if (ch == ')'){
   paren--;
   if (paren == 0){
     args.addElement(new Term(str.substring(i1 + 1, pos)));
     i1 = pos;
     break;}
     }
     else if (str.charAt(pos) == ','){
   if (paren == 1){
     args.addElement(new Term(str.substring(i1 + 1, pos)));
     i1 = pos;
     break;}
              }}}}}
```

# Replacing one of the arguments

```
// replace one of the top level arguments in term
// to make a new term. e.g.  f(a,b).rplarg(1,b) gives f(b,b)
// no error checking done for now.
public Term rplarg(int j, Term narg){
        String tmp = this.opvarname + "(";
        for (int i = 0; i < args.size(); i++){
          if (i == j) {tmp = tmp + narg + ",";}
          else{ Term targ = (Term) args.elementAt(i);
                 tmp = tmp + targ + ",";}}
        //remove extra , at end and add )
        return new Term(tmp.substring(0,tmp.length()-1) + ")")
```

# Representing Substitutions

A Vector of bindings!

```
// need binding <var, term> class first
class Bind{
  public String var;
  public Term  term;
  Bind(String v, Term t){
    var = v; term = t;} }

// a substitution is  a Vector of bindings
// with various methods for adding binding, composing etc.
// how to represent failed subst?
// we use a boolean field isValid
public class Subst{
  public boolean isValid = false;
  public Vector sigma;
  // initialize to ID substitution
  Subst(){
    sigma = new Vector();
    isValid = true;
  }
```

# Substitution Methods

```
public Subst appendBind(String v, Term t){
  // no checking here. simply add at end. ok for matching
  // and when we have normalized already
    sigma.addElement(new Bind(v,t));
  return this;
}

public boolean isBound(String var)

public Term getBind(String v)
public String toString()
```

# Method for matching

```
public Subst match(Term t){
  // returns a sigma such that t matches this term.
  Subst idSub = new Subst();
  return  this.match1(t,idSub);}

public Subst match1(Term t, Subst sigma){
  // assumed that t shares no variable with this term.
  if (t.isvar()){
     return sigma.appendBind(t.opvarname, this);}
  else if (t.opvarname.equals(this.opvarname)){
        for (int i = 0; i < t.args.size(); i++){
          Term targ = (Term) t.args.elementAt(i);
          Term sarg = (Term) this.args.elementAt(i);
          sigma = sarg.match1(targ.applySubst(sigma), sigma);
          if (! sigma.isValid){break;}}}
   else{sigma.isValid=false;};
  return sigma;}
```

Unification code similar, but more complex.

Must **compose** substs instead of appending!

# Representing Rules

```
public class Rule{
  // a rule has lhs and rhs
  public Term lhs;
  public Term rhs;
  // may wish to add other fields like rulenumber later

// a constructor for parsing a string and making a rule
  Rule(String str){
    // assumed to have l -> r with -> as separator

// a method to change names of vars in a rule
  public Rule copy(int num){

// read a set of rules from a file. A static method.
 public static Vector readRules(String fname)

// print rules on terminal
  public static void writeRules(Vector Rules){
```

# Applying a rule

```
// a method that tries to use this rule on input term once.
  public Term rwany(Term t){
    // this rewrites the term ONCE if possible anywhere using
    // return t itself if no rewriting is possible anywhere.
    Subst sig = t.match(lhs);
    if (sig.isValid){
      Term t1 = rhs.applySubst(sig);
      System.out.println(" Rule: " + this +
                         " rewrites " + t + " --> " + t1);
      return t1;}
    else if (t.isvar() | t.isconst())
      return t;
    else{
        for (int i = 0; i < t.args.size(); i++){
          Term targ = (Term) t.args.elementAt(i);
          Term t1 = this.rwany(targ);
          if (!(t1.equals(targ)))
            {return t.rplarg(i,t1);}};
        return t;}}
```

# Computing Normal form

A method on a term.

```
// normalize the term using a set of rules.
public Term norm(Vector rules){
  for (int i = 0; i < rules.size(); i++){
     Rule rule = (Rule) rules.elementAt(i);
     Term ans = rule.rwany(this);
     if (! ans.equals(this)){
         return ans.norm(rules);}};
  return this; // if no rule applies at all
}
```

# Putting it all together

Main.java

```
import java.io.*;
import java.util.*;

public class Main {
  public static void main(String[] agmts){
    Vector rules = Rule.readRules(agmts[0]);
    Rule.writeRules(rules);
    Term t1,t2;
    // t1 = new Term("*(s(0),+(s(0),s(0)))");
    t1 = Term.getTerm();
    t2 = t1.norm(rules);
    System.out.println(t2 + " is normal form of " + t1);
     }
}
```