FORM course: lecture 2

Ben Ruijl

ETH Zurich

Jul 28, 2018

Preprocessor

- Preprocessor instructions are text-based instructions that are executed when the module is compiled
- They start with a #:

```
#define i "2"
#define j "3"
#define k "xx1"
Symbols x`i`,x`j`,x`i``j`,`k`;
Local F = x`i`+x`j`+x`i``j`+`k`;
Print;
.end
```

yields

```
F = xx1 + x23 + x3 + x2;
```

Loops and ...

```
#define MAX "3"
Symbols x1,...,x‘MAX‘;
#do i = 1,‘MAX‘
    L F‘i‘ = (x1+...+x‘i‘)^2;
#enddo
Print;
.end
```

```
F1 =  x1^2;
F2 =  x2^2 + 2*x1*x2 + x1^2;
F3 =  x3^2 + 2*x2*x3 + x2^2 + 2*x1*x3
    + 2*x1*x2 + x1^2;
```

Looping over modules

- Looping modules until a condition is met is a bit tricky
- Use `redefine` to change a preprocessor variable in the next module

```
S x;
CF f;
Local F = f(30);
#do i = 1,1
    id f(x?{>1}) = f(x - 1) + f(x - 2);
    if ( match(f(x?{>1})) );
        redefine i "0";
    endif;
    .sort
#enddo
Print;
.end
```

Homework 1.0

- Substitute $x = 1 - e^y$ into $\ln(1 - x)$ expansion
- First attempt with preprocessor:

```
#define MAX "8"
Symbol x, j;
Local F = sum_(j,1,‘MAX‘,-x^j/j);
.sort
Symbol y(:‘MAX‘),n;  * define cut-off
id x = sum_(j,1,‘MAX‘,-y^j/fac_(j));
Print;
.end
```

Homework 1.5

- Use preprocessor computations between {}
- Use descending do-loop to limit generated powers

```
#define MAX "8"
Symbol x, j;
Local F = sum_(j,1,‘MAX‘,-x^j/j);
.sort
Symbol y(:‘MAX‘),n;
#do i = ‘MAX‘,1,-1
    id x^‘i‘’ = sum_(j,1,{‘MAX‘-‘i‘+1},-y^j/
                    fac_(j))*x^{‘i‘-1};
#enddo
Print;
.end
```

Homework 2.0

- Use a sort to merge terms step by step

```
#define MAX "8"
Symbol x, j;
Local F = sum_(j,1,`MAX`,-x^j/j);
.sort
Symbol y(:`MAX`'),n;
#do i = `MAX`,1,-1
    id x^`i`' = sum_(j,1,{`MAX`-`i`+1},-y^j/
                   fac_(j))*x^{`i`-1};
    .sort: i = `i`;  * label the sort
#enddo
Print;
.end
```

Homework 2.5

- Take the powers of y into account when substituting x
- MAX=50 runs in 0.12 seconds

```
#define MAX "8"
Symbol x, j;
Local F = sum_(j,1,`MAX',-x^j/j);
.sort
Symbol y(:`MAX''),n;
#do i = `MAX',1,-1
    id x^`i'*y^n? = sum_(j,1,{`MAX'-`i'+1}-n,-y^j/
                    fac_(j))*x^{`i'-1}*y^n;
    .sort: i = `i';  * label the sort
#enddo
Print;
.end
```

Dollar variables

- FORM has variables called *dollar variables*
- They are expressions that live in memory
- They are shared between the preprocessor and the algebraic level

```
#$a = 5;  * initialize in compile-time
L F = x^5;

id x^$a = 6;
$a = 7;

Print "%$",$a;
.end
```

Wildcards capturing

- Matches of (ranged) wildcards can be stored in dollar variables

```
S x1,x2;
CF f;
L F = f(1,2,3,4);

id f(x1?$a,x2?$b,?a$c) = 1;
Multiply f($c,f($b),f($a));
Print;
.end
```

```
   F = f(3,4,f(2),f(1));
```

Dollar variables I

- Use dollar variables to uniquely label terms

```
CF f, l;
Local F = f(1)+f(2)+f(3);

#$counter = 0;
Multiply l($counter);
$counter = $counter + 1;
Print;
.end
```

```
F = f(1)*l(1) + f(2)*l(2) + f(3)*l(3)
```

Dollar variables II

- A dollar variable can be used as a preprocessor variable in the next module
- Useful to store global properties

```
Symbols x,y;
Local F = (x+1)^10-(x+3)^6*(x-2)^4;
.sort
#$maxx = 0;
if ( count(x,1) > $maxx );
    $maxx = count_(x,1);
    print " $maxx adjusted to %$",$maxx;
endif;
.sort
#write "The maximum power of x is %$",$maxx
.end
```

Dollar variables III

- Collect global information in one module
- Create dollar ‘table’ in the next module

```
S x, y;
L F = x*y + x^2*y^2 + 2*x^2 + x^3*y;

#$maxpow = 0;
if (count(x,1) > $maxpow) $maxpow = count_(x,1);

Bracket x;
.sort
#do i = 1,‘$maxpow‘
    $a‘i‘ = F[x‘i‘];
#enddo
.end
```

Rational polynomials

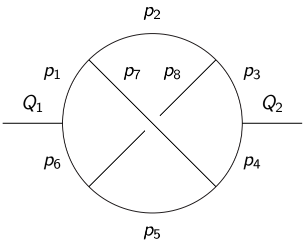
- In many problems such as IBP reductions we need to add multivariate polynomials.
- FORM has the `polyratfun`:

```
S x, y;
CF rat;
polyratfun rat;   * enable polyratfun

L F = rat(y,x) + rat(x,1)*rat(1,y+1);
Print;
.end
```

```
F = rat(x^2 + y^2 + y,x*y + x)
```

Graph automorphisms and id all



```
CF vx(s);  * symmetric function
L F = vx(Q1,p1,p6)*vx(p1,p2,p7)*vx(p2,p3,p8)*
      vx(p3,p4,Q2)*vx(p4,p5,p7)*vx(p5,p6,p8);

id all vx(Q1?,p1?,p6?)*vx(p1?,p2?,p7?)*vx(p2?,p3?,p8?)*
      vx(p3?,p4?,Q2?)*vx(p4?,p5?,p7?)*vx(p5?,p6?,p8?) =
      map(Q1,Q2,p1,p2,p3,p4,p5,p6,p7,p8);
```

Automorphisms

This gives:

```
F =
+ map(Q1,Q2,p1,p2,p3,p4,p5,p6,p7,p8)
+ map(Q1,Q2,p1,p7,p4,p3,p8,p6,p2,p5)
+ map(Q1,Q2,p6,p5,p4,p3,p2,p1,p8,p7)
+ map(Q1,Q2,p6,p8,p3,p4,p7,p1,p5,p2)
+ map(Q2,Q1,p3,p2,p1,p6,p5,p4,p8,p7)
+ map(Q2,Q1,p3,p8,p6,p1,p7,p4,p2,p5)
+ map(Q2,Q1,p4,p5,p6,p1,p2,p3,p7,p8)
+ map(Q2,Q1,p4,p7,p1,p6,p8,p3,p5,p2)
;
```

Example: term-local unique counter

```
CF fnum,vx,vxx;
L F = vx(1,2)*vx(3,4)*vx(5,6);

Multiply fnum(1);
repeat id vx(?a)*fnum(n?) = vxx(n,?a)*fnum(n+1);
id vxx(?a) = vx(?a);
.end
```

Example: check if graph is connected

```
repeat id vx(?a,p?,?b)*vx(?c,-p?,?d) =
                vx(?a,?b,?c,?d);
if (count(vx,1) == 1);
     Print "Connected!";
else;
     Print "Not connected";
endif;
```

Example: finding cutvertices

A cutvertex makes a graph disconnected if removed

```
Multiply f;
repeat id vx(?a)*f(x?) = f(x*vx(?a));
argument f;
    id all vx(?a) = vxx(?a);  * select a vertex
    repeat id vx(?a,p?,?b)*vx(?c,-p?,?d) =
                   vx(?a,?b,?c,?d);
    if (count(vx,1) < 2) Discard;
    id vx(?a) = 1;
endargument;
splitarg f;  * f(x+x1) -> f(x,x1)
```

Expression simplification

- Reduce number of operations of polynomial evaluation
- Much faster (factor 10) polynomial sampling

```
S    a,b,c,d,e,f,g,h,i,j,k,l,m,n;
L    G = (4*a^4+b+c+d + i^4 + g*n^3)^10
         + (a*h + e + f*i*j + g + h)^8
         + (i + j + k + l + m + n)^12;
Format O4,saIter=300;  * use 300 iterations
.sort
#optimize G
#write "Number of operations in output: `optimvalue_'"
#clearoptimize
.end
```

reFORM

- A FORM replacement in early development
- Fixes some of FORM's limitations
- Should be easier to use

```
expr F = f(2+y,x*y);

for $i in 1..5 {
    apply {
        id f($i+x?,x?*y?) = f(x?);
    }
}
print;
```

Exercises

- Write a FORM program that reads unique topologies from an expression F and stores them in dollar variables
- Use this 'database' to find all unique graphs (up to isomorphisms) in an expression G