Symbolica

Introduction to FORM: part 2

Ben Ruijl

July 17 - 19, 2023

Ruijl Research

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 ... operator

```
#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
```

Preprocessor exercise

- $a = \ln(1 - x)$ expansion is $a = -x - \frac{x^2}{2} - \frac{x^3}{3} - \ldots$
- $b = 1 - e^y$ expansion is $b = -y - \frac{y^2}{2} - \frac{y^3}{6} - \ldots$
- Write a FORM program that substitutes x in a by b for a fixed expansion depth `MAX`

Exercise 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
```

Exercise 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
```

Exercise 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
```

Exercise 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
```

Expression optimisation

- Reduce number of operations of polynomial evaluation
- Much faster polynomial sampling

```
S x,y,z;
L F = (x*y+6*x+z^2)
 *(x^2+y^2+z^2+1);

Format O4;
.sort
#Optimize F
#write "%O";
Print F;
.end
```

```
Z1_=y + 6;
Z2_=z^2;
Z3_=Z1_*Z2_;
Z4_=x*Z1_;
Z4_=Z2_ + Z4_;
Z4_=x*Z4_;
Z1_=y*Z1_;
Z1_=1 + Z1_;
Z1_=y*Z1_;
Z1_=Z4_ + Z3_ + 6 + Z1_;
Z1_=x*Z1_;
Z3_=y^2;
Z3_=Z2_ + 1 + Z3_;
Z2_=Z3_*Z2_;
F=Z1_ + Z2_;
```

Expression optimisation

- Reduce number of operations of polynomial evaluation
- Much faster polynomial sampling

```
S x,y,z;
L F = (x*y+6*x+z^2)
 *(x^2+y^2+z^2+1);

Format O4;
.sort
#Optimize F
#write "%O";
Print F;
.end
```

```
Z1_=y + 6;
Z2_=z^2;
Z3_=Z1_*Z2_;
Z4_=x*Z1_;
Z4_=Z2_ + Z4_;
Z4_=x*Z4_;
Z1_=y*Z1_;
Z1_=1 + Z1_;
Z1_=y*Z1_;
Z1_=Z4_ + Z3_ + 6 + Z1_;
Z1_=x*Z1_;
Z3_=y^2;
Z3_=Z2_ + 1 + Z3_;
Z2_=Z3_*Z2_;
F=Z1_ + Z2_;
```

Extra symbols I

- Any FORM expression can be converted to a symbol using `extrasymbols_`
- Effectively creates a map from a key to a symbol

```
Auto S x;
CF f;

L F = f(x1)*f(x2)*f(x1*x2) + x1*f(x2);

#define start "{`extrasymbols_'+1}"
argtoextrasymbol tonumber, f;
.sort:collect;
```

gives

```
   F = f(1)*f(2)*f(3) + x1*f(2);
```

Extra symbols II

- Iterate over all new 'extra' symbols and creates new expressions:

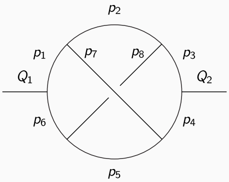
```
#define end "`extrasymbols_'"
#do i=`start`,`end'
    L F`i' = extrasymbol_(`i');
#enddo
```

yields:

```
  F1 = x1;
  F2 = x2;
  F3 = x1*x2;

```

Applications




```
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);
```

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
```

yields `vx(1,1,2)*vx(2,3,4)*vx(3,5,6)`

Exercises I

```
V p, k1,...,k4;
I mu1,...,mu11,nu1,nu2;
CF vx;

L F = vx(-p,p-k1,k1,nu1,mu1,mu8)*
      vx(-k1,k2,k1-k2,mu1,mu2,mu9)*
      vx(-k2,k3,k2-k3,mu2,mu3,mu10)*
      vx(-k3,k4,k3-k4,mu3,mu4,mu11)*
      vx(-k4,p,k4-p,mu4,nu2,mu5)*
      vx(-k4+p,-k3+k4,k3-p,mu5,mu11,mu6)*
      vx(-k3+p,-k2+k3,k2-p,mu6,mu10,mu7)*
      vx(-k2+p,-k1+k2,k1-p,mu7,mu9,mu8);
```

- Every vertex is a triple-gluon vertex
- Implement Feynman rules
- Use smart sorts to make the code run faster!

Large exercise II

- UV expand the one-loop photon self-energy
- Take limit of $p \to 0$: rescale p with λ
- Compute counterterm using one-loop IBP for massive graphs

Good luck!