Advanced Maple

WHAT YOU WONT FIND IN MY THESIS

STEVEN E. THORNTON

Table of Contents

Functions

Functions

```
procName := proc(parameterSequence) :: returnType;

    local localSequence;
    global globalSequence;
    option optionSequence;
    description descriptionSequence;
    uses usesSequence;

    statementSequence;

end proc:
```

Default Argument Values

```
Log := proc(z, K := 0)

    return log(z) + 2*Pi*I*K;

end proc:
```

Default Argument Values

```
Log := proc(z, K := 0, base := 10)

    return log[base](z) + 2*Pi*I*K/log(base);

end proc:
```

Optional Arguments

```
Log := proc(z, {K := 0, base := 10})
    return log[base](z) + 2*Pi*I*K/log(base);
end proc:
```

Type Checking

```
myGCD := proc(p1, p2, v)

    return gcd(p1, p2);

end proc:
```

Type Checking

```
myGCD := proc(p1::polynom, p2::polynom, v::symbol)

    return gcd(p1, p2);

end proc:
```

Type Checking

```
myGCD := proc(p1::polynom(integer),
              p2::polynom(integer),
              v::symbol, $)

    return gcd(p1, p2);

end proc:
```

Type Checking

```
myGCD := proc(p1::depends(polynom(integer, v)),
              p2::depends(polynom(integer, v)),
              v::symbol, $)

    return gcd(p1, p2);

end proc:
```

Type Checking

```
myGCD := proc(p1::depends(polynom(integer, v)),
              p2::depends(polynom(integer, v)),
              v::symbol, $)

    local d1, d2;

    d1, d2 := degree(p1), degree(p2);

    if d1 > 10 or d2 > 10 then
        error "myGCD doesn't work for polynomials of degree > 10";
    end if;

    return gcd(p1, p2);

end proc:
```

Type Checking

```
myGCD := proc(p1::depends(polynom(integer, v)),
              p2::depends(polynom(integer, v)),
              v::symbol, $)::polynom(integer, v);

    local d1::nonnegint, d2::nonnegint;

    d1, d2 := degree(p1), degree(p2);

    if d1 > 10 or d2 > 10 then
        error "myGCD doesn't work for polynomials of degree > 10";
    end if;

    return gcd(p1, p2);

end proc:

kernelopts(assertlevel = 2);     # ONLY USE FOR TESTING
```

TypeTools

The TypeTools package allows you to extend the known types within Maple.

```
TypeTools[AddType](pair, proc(expr, elementtype := anything)
    type(expr, [elementtype, elementtype])
end proc):


TypeTools[AddType](TRDring, proc(expr)
    RegularChains:-TRDis_polynomial_ring(expr);
end proc):
```

Modules

What are Modules?

Modules are a highly versatile type within Maple. They can be used to:

- Create packages (`LinearAlgebra`, `RegularChains`, etc.)
- Provide object oriented programming techniques
- Write functions with subroutines local to that function
- Keep your code organized and modular

Syntax

```
moduleName := module()

    export eseq;
    local lseq;
    global gseq;
    option optseq;
    description dseq;
    uses usesSeq;

    statementSequence

end module:
```

Module as a Function

```
myGCD := module()

end module:
```

Module as a Function

```
myGCD := module()

    export ModuleApply;

end module:
```

Module as a Function

```
myGCD := module()

    export ModuleApply;

    ModuleApply := proc(p1, p2, v)
        return gcd(p1, p2);
    end proc;

end module:
```

Module as a Function

```
myGCD := module()

    export ModuleApply;

    ModuleApply := proc(p1::depends(polynom(integer, v)),
                        p2::depends(polynom(integer, v)),
                        v::symbol, $)::polynom(integer, v);

        return gcd(p1, p2);

    end proc;

end module:
```

Module as a Function

```
myGCD := module()

    export ModuleApply;

    local checkInput,
          implementation;

    ModuleApply := proc()
        local p1, p2, v;
        p1, p2, v := checkInput(args);
        return implementation(p1, p2, v);
    end proc;

end module:
```

Module as a Function

```
checkInput := proc(p1::depends(polynom(integer, v)),
                   p2::depends(polynom(integer, v)),
                   v::symbol, $)

    local d1::nonnegint, d2::nonnegint;

    d1, d2 := degree(p1), degree(p2);

    if d1 > 10 or d2 > 10 then
        error cat(procname, " doesn't work for polynomials of
degree > 10");
    end if;

    return p1, p2, v;

end proc;
```

Module as a Function

```
implementation := proc(p1, p2, v)
    return gcd(p1, p2);
end proc;
```

Packages

Packages

```
with(RegularChains);

R := PolynomialRing([x, y, z]);

sys := [x^2+y+z-1, y^2+x+z-1, z^2+x+y-1];

rc := Triangularize(sys, R);

Display(rc, R);
```

Creating a Package

```
IrregularChains := module()

    option package;

end module:
```

Creating a Package

```
IrregularChains := module()

    option package;

    export PolynomialRing;

end module:
```

Creating a Package

```
IrregularChains := module()

    option package;

    export PolynomialRing;

    PolynomialRing := proc(l)
        return table(['variables' = l]);
    end proc;

end module:
```

Creating a Package

```
IrregularChains := module()

    option package;

    export PolynomialRing,
           printPolynomialRing;

    local getRingVariables;

    PolynomialRing := proc(l)
        return table(['variables' = l]);
    end proc;

end module:
```

Creating a Package

```
IrregularChains := module()

    option package;

    export PolynomialRing,
           printPolynomialRing;

    local getRingVariables;

    # ...

    printPolynomialRing := proc(R)
        print(getRingVariables(R));
    end proc;

    getRingVariables := proc(R)
        return R['variables'];
    end proc;
```

Creating a Package

```
IrregularChains := module()

    option package;

    export PolynomialRing,
           printPolynomialRing,
           reverseVariableOrder;

    local getRingVariables;

    uses ListTools;

    # ...

    reverseVariableOrder := proc(R)
        return PolynomialRing(Reverse(getRingVars(R)));
    end proc;
```

Creating a Package

```
with(IrregularChains);

R := PolynomialRing([x,y,z]):

printPolynomialRing(R):

R2 := reverseVariableOrder(R):

printPolynomialRing(R2):
```

Objects

Modules as Objects

```
Car := proc(model, car, $)
    module()
        export getNumWheels,
               setNumWheels,
               getHighwayMPG,
               setHighwayMPG,
               ModulePrint;

        local modelName := model,
              carType := car,
              numWheels := 4,
              highwayMPG := 34;

        # ...

    end module;
end proc:
```

Mint

File Organization

Makefile