gfun[rectoproc] - convert a recurrence into a function

Calling Sequence

rectoproc(**eqns**,**u(n)**, **<remember>**, **<list>**,**<params=[a,b,...]>**,**<options>**)

Parameters

`eqns`	–	single equation or set of equations
`u,n`	–	name and index of recurrence
`remember`	–	(optional)
`list`	–	(optional)
`params = [a,b,...]`	–	(optional) names a,b,... for the arguments
`evalfun = fname`	–	(optional) a function applied at each iteration
`preargs=[c,d,...]`	–	(optional) first arguments to this function
`postargs=[e,f,...]`	–	(optional) last arguments to this function
`evalinicond`	–	(optional)
`whilecond=boolean_expr`	–	(optional) while condition
`errorcond=boolean_expr`	–	(optional) error condition
`index`	–	(optional)
`rhs=expression`	–	(optional)
`copyright=string`	–	(optional) copyright string to be added to the options
`extralocal=[g,h,...]`	–	(optional) names for extra local variables
`nosymbolsubs`	–	(optional)
`plain`	–	(optional)

Description

The procedure outputs a Maple procedure that, given a non negative integer **n** as input, gives the **n**-th term **u(n)** of the linear recurrence.

If the optional argument **remember** is supplied, the procedure returned will use **option remember**.

If the optional argument **list** is supplied, the procedure returned will compute the list **[u(0),...,u(n)]**. In this case it will run in a linear number of arithmetic operations.

One of the optional arguments **remember** or **list** should be given each time one needs a large number of values of the sequence. When the first terms of the recurrence are not explicitly supplied, they are represented symbolically.

If the optional argument **plain** is supplied, the procedure returned will "unroll" the recurrence relation using a plain loop, even in cases where more efficient algorithms are supported. This is allows to use the resulting procedure with evalhf, the CodeGeneration package, and other functions that do not support all Maple objects or language features.

If the optional argument **params=[a,b,...]** is given then the procedure will take as input **n**, **a**, **b**,...

When the coefficients of the recurrence are nonconstant polynomials the returned procedure will run in quasi-linear time and space if the remember option is not specified.

When the coefficients are constant, the procedure will run in logarithmic number of arithmetic operations and without wasting extra space if the remember option is not specified.

If the optional argument **evalfun=fname** is given, then the specified procedure is used in the generated code as an evaluation rule. Extra arguments can be passed to the procedure with options **preargs=[c,d,...]** and **postargs=[e,f,...]**. Names declared in **preargs** and **postargs** may appear in the argument sequence of the generated procedure if they are also declared with the option **params**. The function is mapped over initial conditions and it is evaluated before procedure generation if the option **evalinicond** is supplied.

The option **whilecond=boolean_expr** defines a condition that is checked at each iteration of the loop of the generated procedure. The condition is represented by a boolean expression that may be function of **n**, **u(n-k)** for any positive integer **k** and function of the names declared in **params**, **preargs** and **postargs**. Execution stops when the condition turns true. This option does not impact initial conditions.

The option **errorcond=boolean_expr** defines a condition that is checked at each iteration of the loop of the generated procedure. The condition is represented by a boolean expression that may be function of **n**, **u(n-k)** for any positive integer **k** in **0..ord** - where **ord** is the order of the recurrence - and function of the names declared in **params**, **preargs** and **postargs**. An error is thrown when the condition turns true.

The option **extralocal=[g,h,...]** allows to declare and initialize extra local variables. **g**, **h**, ... must be either symbols or equations in the form **symbol=expression**, in which case the expression is used for initialization.

The option **nosymbolsubs** disables the name substitution that occurs for the parameters (declared with the option **params**). This means that the symbols that are used in the generated procedures are the same as the symbols used in the input recurrence. By default, the symbols that are used in the generated procedures are gathered in the global table **gfun/rectoproc/symbol**.

The option **index** makes the generated procedure return the list **[n,u(n)]**. This is useful in conjunction with **whilecond**.

The option **rhs=expression** allows to specify a right hand side of the recurrence that does not have to be a polynomial. The right hand side may be function of **n** and of the names declared in the options **params**, **preargs** and **postargs**.

Examples

Fibonacci numbers

We can create different types of programs to generate Fibonacci numbers that are tailored to certain memory or time requirements.

The most obvious procedure is recursive and "remembers" values that are already computed.

```
> fiborec:={f(i)=f(i-1)+f(i-2),f(0)=1,f(1)=1};
```

$$\{f(0)=1, f(1)=1, f(i)=f(i-1)+f(i-2)\} \tag{2.1}$$

```
> with(gfun):
  fib1:=rectoproc(fiborec,f(i),remember);
```

```
proc(n::nonnegint)                                                    (2.2)
    option remember;
    procname(-2+n)+procname(-1+n)
end proc
```

```
> fib1(100);
```

$$573147844013817084101 \tag{2.3}$$

To create a program which computes and lists the first **n** terms of the recurrence, we use the list option.

```
> fib2:=rectoproc(fiborec,f(i),list);
```

```
proc(n::nonnegint)                                                    (2.4)
    local i1, loc;
    loc[0]:=1;
    loc[1]:=1;
    for i1 to -1+n do loc[i1+1]:=loc[-1+i1]+loc[i1] end do;
    [seq(loc[i1], i1=0..n)]
end proc
```

```
> fib2(10);
```

$$[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89] \tag{2.5}$$

To create a program that is more space conscious we avoid the remember option. In this case the program exploits the fact that the recurrence has constant coefficients for extra efficiency.

```
> fib3:=rectoproc(fiborec,f(i));
```

```
proc(n::nonnegint)                                                    (2.6)
    local i1, loc0, loc1, loc2, tmp2, tmp1, i2;
    if n <= 44 then
        loc0:=1;
        loc1:=1;
        if n=0 then
            return loc0
        else
            for i1 to -1+n do loc2:=loc0+loc1; loc0:=loc1; loc1:=loc2 end do
        end if;
        loc1
    else
        tmp2:=`Matrix(2, 2, {(1, 1) = 1, (1, 2) = 1, (2, 1) = 1, (2, 2) = 0})`;
        tmp1:=`Vector(2, {(1) = 1, (2) = 1})`;
        i2:=convert(-1+n, base, 2);
        if (`Vector(2, {(1) = 1, (2) = 1})`)[1]=1 then
            tmp1:=`Vector(2, {(1) = 1, (2) = 1})`:=
            `Vector(2, {(1) = `+`(`-`(1), n), base, 2, (2) = 1})`
        end if;
```

```
        for i1 in subsop(1 = ( ), `Vector(2, {(1) = 1, (2) = 1})`) do
            tmp2 := `LinearAlgebra:-MatrixMatrixMultiply`(tmp2, tmp2);
            if i1 = 1 then
                tmp1 := `Vector(2, {(1) = 1, (2) = 1})` := MatrixVectorMultiply(tmp2,
                tmp1 := `Vector(2, {(1) = 1, (2) = 1})`)
            end if
        end do;
        (tmp1 := `Vector(2, {(1) = 1, (2) = 1})`)[(`Vector(2, {(1) = 1, (2) = 1})`)[1] = 1]
    end if
end proc
```

```
> fib3(100);
```

$$57314784401381708 4101 \tag{2.7}$$

```
> fib3(1000);
```

$$70330367711422815821835254877183549770181269836358732742604905087154537118\backslash \\ 19693357974224949456261173348775044924176599108818636326545022364710601\backslash \\ 20533741212738673391111981393731255987676900919022452453234035 01 \tag{2.8}$$

An example of right-hand side: nested procedures

This example shows how terms of nested recurrences can be computed. The first sequence is first converted into a procedure:

```
> rec_u:={u(n+2)*(n^2+n+1)+u(n+1)*(n-2)+u(n)*n,u(0)=1,u(1)=2};
```

$$\{u(n+2)\,(n^2+n+1)+u(n+1)\,(-2+n)+u(n)\,n, u(0)=1, u(1)=2\} \tag{2.9}$$

```
> u_n := rectoproc(rec_u,u(n),remember);
```

```
proc(n::nonnegint)                                                    (2.10)
    option remember;
    (2*procname(-2+n)+4*procname(-1+n)-(procname(-2+n)
     +procname(-1+n))*n)/(3+(-3+n)*n)
end proc
```

The second sequence is defined by

```
> rec_v:=v(n+3)*(n+4)+v(n+1)*(n^2+2*n)=u(n+1)-n*u(n);
```

$$v(n+3)\,(n+4)+v(n+1)\,(n^2+2\,n)=u(n+1)-u(n)\,n \tag{2.11}$$

```
> ini_v:={v(0)=1,v(1)=2,v(2)=3};
```

$$\{v(0)=1, v(1)=2, v(2)=3\} \tag{2.12}$$

The procedure computing **v(n)** is now generated

```
> v_n:=rectoproc({op(1,rec_v)}union ini_v,v(n),rhs=subs(u=u_n,op(2,
  rec_v)),list);
```

```
proc(n::nonnegint)                                                    (2.13)
    local i1, loc;
    loc[0] := 1;
    loc[1] := 2;
    loc[2] := 3;
    for i1 from 2 to -1+n do
```

```
        loc[il + 1] := (u_n( - 1 + il) - u_n( - 2 + il) * ( - 2 + il) - (3 + ( - 3
        + il) * (il + 1)) * loc[ - 1 + il]) / (il + 2)
    end do;
    [seq(loc[il], il = 0 ..n)]
end proc
```

Computation of 10 first terms:

```
> v_n(9);
```

$$\left[1, 2, 3, \frac{1}{2}, -\frac{7}{5}, -\frac{17}{9}, \frac{125}{49}, \frac{6803}{1092}, -\frac{169567}{17199}, -\frac{423697}{14105}\right] \tag{2.14}$$

Applying a function at each iteration

We illustrate several ways to compute numerical values of the sequence defined by the following recurrence:

```
> rec := { u(n+2)*(n+1) + u(n)*(n^2+1),u(0)=sin(1),u(1)=cos(1)};
```

$$\{u(n+2)\ (n+1) + u(n)\ (n^2+1), u(0) = \sin(1), u(1) = \cos(1)\} \tag{2.15}$$

If hardware floating point numbers give a sufficient accuracy, then it is best to produce the procedure and then call evalhf on it:

```
> p:=subsop(4=NULL,rectoproc(rec,u(n),'plain'));
```

```
proc(n::nonnegint)                                                    (2.16)
    local il, loc0, loc1, loc2;
    loc0 := sin(1);
    loc1 := cos(1);
    if n = 0 then
        return loc0
    else
        for il to  - 1 + n do
            loc2 := - (5 + ( - 3 + il) * (il + 1)) * loc0 / il; loc0 := loc1; loc1 := loc2
        end do
    end if;
    loc1
end proc
```

```
> evalhf(p(100));
```

$$5.27739153869639746\ 10^{76} \tag{2.17}$$

If more precision is required, then evalf can be applied at each iteration using

```
> p2:=rectoproc(rec,u(n),evalfun='evalf');
```

```
proc(n::nonnegint)                                                    (2.18)
    local il, loc0, loc1, loc2;
    loc0 := evalf(sin(1));
    loc1 := evalf(cos(1));
    if n = 0 then
        return loc0
    else
```

```
        for i1 to -1+n do
            loc2 := evalf(-(5+(-3+i1)*(i1+1))*loc0/i1); loc0 := loc1; loc1 :=
            loc2
        end do
    end if;
    loc1
end proc
```

```
> Digits:=30: p2(100);
```

$$5.27739153869639769206242956896\ 10^{76} \tag{2.19}$$

It is also possible to give the precision as an argument:

```
> p3 := rectoproc(rec,u(n),evalfun='evalf',params=[d],postargs=[d])
  ;
```

```
proc(n::nonnegint, b1)
    local i1, loc0, loc1, loc2;
    loc0 := evalf(sin(1), b1);
    loc1 := evalf(cos(1), b1);
    if n = 0 then
        return loc0
    else
        for i1 to -1+n do
            loc2 := evalf(-(5+(-3+i1)*(i1+1))*loc0
            /i1, b1);
            loc0 := loc1;
            loc1 := loc2
        end do
    end if;
    loc1
end proc
```

(2.20)

```
> p3(100,40);
```

$$5.27739153869639769206242956899017008682 1\ 10^{76} \tag{2.21}$$

Extra local variables

The option **extralocal** is useful when the values of some parameters are not known until the procedure is executed. In the example below, the recurrence is provided with generic intial conditions. The values of these initial conditions are computed at execution-time.

```
> rec := { u(n)*n + u(n+2),u(0)=A,u(1)=B}:
  p := rectoproc(rec,u(n),params=[p],extralocal=[A='f'(p),B='g'(A,
  p)]);
```

```
proc(n::nonnegint, b1)
    local i1, loc0, loc1, loc2, tmp2, tmp1, i2, xloc1, xloc2;
    xloc1 := f(b1);
    xloc2 := g(xloc1, b1);
    if n <= 20 then
        loc0 := xloc1;
        loc1 := xloc2;
```

(2.22)

```
        if n = 0 then
            return loc0
        else
            for i1 to -1 + n do
                loc2 := -(-1 + i1) * loc0; loc0 := loc1; loc1 := loc2
            end do
        end if;
        loc1
    else
        tmp1 := `gfun/rectoproc/binsplit`(['ndmatrix'(Matrix([[0, 1], [-i2, 0]]), 1), i2,
        0, n, matrix_ring(ad, pr, ze,
        ndmatrix(`Matrix(2, 2, {}, storage = empty, shape = [identity])`, 1)),
        expected_entry_size], `Vector(2, {(1) = u(0), (2) = ()(1)})`);
        tmp1 := subs({expected_entry_size = xloc1, u = xloc2}, tmp1);
        tmp1
    end if
end proc
```

See Also

gfun, remember, nth_term, diffeqtoproc