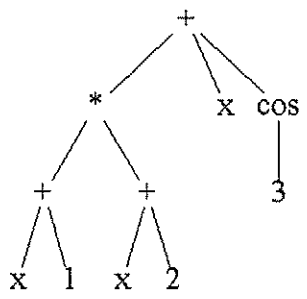
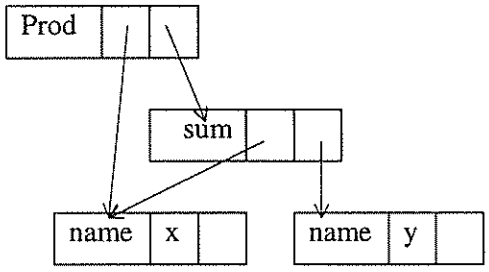
Une autre façon de visualiser cette structure consiste à noter les expressions sous la forme d'arbres[4] :



Cette visualisation est probablement la manière la plus efficace de comprendre la structure d'une expression algébrique.

Quel que soit le mode de représentation adopté, il faut retenir le fait que toute expression mathématique s'obtient en appliquant une fonction à une liste d'arguments pouvant être des nombres, des variables symboliques, ou des sous-expressions, elles-mêmes définies en utilisant d'autres fonctions, et que tout logiciel de calcul symbolique est capable d'accéder facilement à ces différents éléments, et donc de déterminer le type de l'expression manipulée.

De plus, on sait parfaitement bien représenter ce type de structure dans la mémoire d'un ordinateur. L'idée est que chaque élément est décrit par une liste donnant la nature de l'expression associée ainsi que la liste des arguments. Les "atomes" c'est à dire les éléments de base des expressions comme x, 10 ou π peuvent eux-mêmes être représentés par des descripteurs permettant de connaître leur valeur, leur nom, et éventuellement certaines de leurs propriétés[5]. La représentation interne d'une expression comme $x \cdot (x + y)$ pourrait être quelque chose comme :



On voit ici deux types de descripteurs : des descripteurs d'expressions (prod, sum) et des descripteurs de variables (name). Les flèches figurant dans le diagramme précédent sont remplacées par des adresses mémoire indiquant l'emplacement du descripteur associé à chaque argument. Les descripteurs associés aux variables x et y comportent ici trois champs. Le premier donne le type de l'objet représenté (ce sont des variables), le second permet de connaître le nom, le troisième permet de préciser la valeur de la variable lorsque celle-ci est affectée. L'absence de pointeur dans cette troisième case des descripteurs des variables x et y correspond au fait que ces variables n'ont pas reçu de valeur.

---

[4] Il est possible de générer automatiquement ce type de représentation avec Maple en utilisant la fonction **tree** dont l'écriture est décrite dans le livre *Premiers pas en Maple*, paru aux éditions Vuibert.

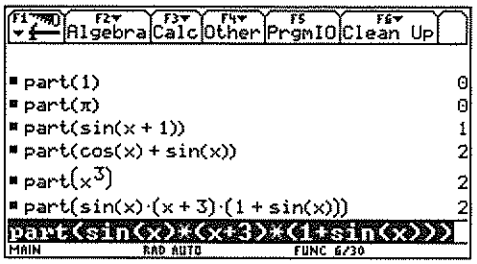
[5] Il existe différentes façons de représenter une liste en mémoire. Nous n'entrerons pas ici dans le détail de cette représentation. On pourra aussi se reporter au livre *Introduction to Maple*, André Heck, Springer-Verlag pour avoir plus de précision sur les méthodes utilisées par Maple pour la représentation interne des expressions.

La fonction part

Si l'on utilise une TI-89, une TI-92 Plus, ou une Voyage 200 il est possible d'accéder, du moins indirectement, à ces différents éléments en utilisant la fonction **part**.

- Utilisée avec un seul argument, elle donne le nombre de sous-expressions composant l'expression.
- Pour avoir accès à la fonction permettant de construire l'expression (celle qui figure en haut de l'arbre...), on utilise **part** avec un deuxième argument égal à 0.
- Pour avoir accès aux sous-expressions utilisées pour former l'expression, on utilise **part** avec un second argument égal au numéro de la sous-expression.

En voici quelques exemples :



Comme on peut le voir, cette fonction ne suit pas exactement le modèle précédent lorsqu'elle donne les éléments d'une expression.

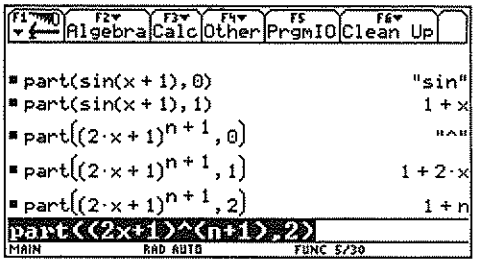
De manière interne, une somme peut avoir un nombre quelconque de termes, il en est de même pour un produit...

Dans le dernier calcul, on utilise l'expression $\sin(x)(x+3)(1+\sin(x))$, qui est le produit des trois facteurs $\sin(x)$, $(x+3)$, et $(1+\sin(x))$.

On pourrait donc s'attendre à obtenir la valeur 3 lors du calcul de `part(sin(x)(x+3)(1+sin(x)).`

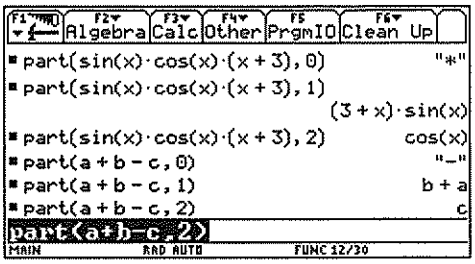
Mais en fait, avec la fonction **part**, les différents termes ou facteurs seront regroupés de manière à toujours se ramener à une somme de deux termes ou à un produit de deux facteurs.

Voici à présent quelques exemples "d'extraction" de l'opérateur utilisé pour construire une expression, et des sous-expressions auxquelles il s'applique :



Une fois encore, il faut insister sur le fait que la fonction **part** ne traduit pas toujours exactement la véritable structure interne des expressions.

Nous l'avons déjà signalé pour les sommes de plusieurs termes, ou avec les produits de plusieurs facteurs, mais c'est aussi par exemple le cas avec des expressions comme $a+b-c$. Une expression de ce type est traitée en interne comme une somme de trois termes : $a, b, -c$, alors que la fonction y verra la différence entre $a+b$ et c comme le montre l'écran suivant :

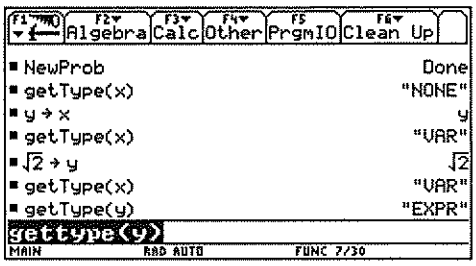


N.B. Cette particularité de la fonction **part** a pour avantage de toujours se ramener à des produits de deux facteurs, des sommes de deux termes, ... mais elle complique par contre l'écriture de certains programmes où il est souhaitable de pouvoir disposer directement de la liste de tous les facteurs d'un produit ou de tous les termes d'une somme.

On doit alors écrire une fonction utilisant des appels récursifs à la fonction **part** pour construire cette liste.

Sur les TI-89, TI-92 Plus et Voyage 200, on dispose également de la fonction **getType** qui permet de savoir quelle est la nature du contenu d'une variable.

Par exemple, on obtient "NONE" si la variable n'a pas de valeur, "NUM" si c'est une valeur entière, ou un nombre rationnel, "VAR" si c'est le nom d'une autre variable, "EXPR" si c'est une expression faisant intervenir une somme, un produit, ou toute autre fonction.



Comme on peut le voir dans l'écran ci-dessus, cette fonction ne permet pas de distinguer entre une expression faisant intervenir des variables symboliques, comme $\sin(x)+x$, et une expression numérique constante nécessitant l'utilisation de fonctions particulières pour son expression. Par exemple $\ln(2)$ ou encore $\sqrt{5}$ sont de type "EXPR".

Les spécialistes pourront également remarquer que cette fonction a un caractère un peu particulier. Généralement, lorsque l'on utilise une fonction f avec un argument arg la TI-89, ou la Voyage 200, commence par évaluer la valeur de arg, puis transmet la valeur obtenue à la fonction f.

Dans l'exemple précédent, le calcul de $f(x)$ serait équivalent à celui de $f(y)$, et donc de $f\left(\sqrt{2}\right)$.

Ici c'est bien le type de x que l'on a obtenu, et non celui de y.

Nous allons à présent voir comment utiliser ces outils pour mettre en évidence certains aspects du fonctionnement des logiciels de calcul formel.

Un exemple complet d'utilisation de part : la dérivation

Non, n'ayez pas d'inquiétude, la TI-89 ou la Voyage 200 sont bien capables de calculer une dérivée.

Nous allons cependant voir comment lui apprendre à le faire, afin d'avoir un exemple d'utilisation de **part** un peu plus complet que celui de la recherche d'un ensemble de définition déjà présent dans le manuel.

Pour cela, il faut prendre le temps de comprendre comment on procède à la main.

- Si l'on a une expression qui ne comporte qu'un seul élément simple, comme x, y ou 10, on regarde si cette expression est ou non égale à la variable par rapport à laquelle on souhaite dériver. Si c'est le cas, la dérivée est égale à 1, sinon la dérivée est égale à 0.

- Si l'on a une expression construite en appliquant une fonction à une expression, comme par exemple $\sin(x)$ ou encore $\ln\left(x^2+1\right)$, on doit appliquer la formule de dérivation d'une expression du type $f(u(x))$ qui est $f'(u(x))\cdot u'(x)$.
  Le calcul de f' sera fait par simple lecture dans la table des dérivées usuelles, le calcul de $u'(x)$ sera fait en appliquant la méthode de dérivation que nous sommes en train de décrire à $u(x)$.

- Si l'on a une somme ou une différence (ces deux notions sont distinctes pour la fonction **part**, contrairement à ce qui se passe en interne) alors on peut appliquer la formule de dérivation d'une somme ou d'une différence. Le calcul de la dérivée de chaque terme se fait en appliquant la méthode que nous sommes en train de décrire.

- Si l'on a un produit, on dispose de la formule $(uv)' = u'v + uv'$.
  (Le cas du produit par une constante est un cas particulier du précédent. Dans la mesure où nous ne cherchons pas à écrire une programme destiné à afficher les étapes pas à pas et travaillant de la même façon qu'un étudiant, il n'est pas utile de le traiter.)

- Si l'on a un quotient on dispose de la formule $\left(\frac{u}{v}\right)' = \frac{u'v - uv'}{v^2}$.
  (Le cas d'un inverse, c'est à dire le cas où le numérateur est égal à 1, est un cas particulier. Pour la même raison que dans le cas du produit par une constante, il est inutile de le traiter.)

- Si l'on a une puissance, la méthode à utiliser dépend du type de l'exposant et du type de la base. Dans un calcul manuel, on distinguerait les cas :
  $f(x) = u(x)^{\alpha}$, avec $f'(x) = \alpha u'(x) u(x)^{\alpha-1}$.
  $f(x) = e^x$, avec $f'(x) = e^x$, et plus généralement $f(x) = a^x = e^{x\ln(a)}$ avec $f'(x) = \ln(a) a^x$
  $f(x) = u(x)^{v(x)} = e^{v(x)\ln(u(x))}$, avec $f'(x) = \left(\frac{u'(x)\cdot v(x)}{u(x)} + \ln(u(x))\cdot v'(x)\right) u(x)^{v(x)}$
  Cette dernière formule n'est sans doute pas connue des élèves. Nous pouvons par contre parfaitement décider de l'utiliser systématiquement afin d'éviter des tests sur la nature de l'exposant et de la base. Elle se simplifiera lorsque l'un ou l'autre sera constant.

- Faut-il prévoir d'autres cas ?
  Nous pouvons pour l'instant décider de ne pas les traiter, quitte à revenir sur ce problème si cela est vraiment nécessaire.
  Dans cette rubrique pourrait par exemple figurer le cas d'une expression faisant intervenir une intégrale, comme $f(x) = \int_x^1 e^{-t^2}\,dt$.

Il existe des techniques de programmation (programmation orientée objet) qui permettent une prise en compte automatique de nouvelles méthodes, sans avoir besoin de modifier la fonction principale.

Il n'est cependant pas possible de développer ce type de méthodes dans le cadre de ce document, nous allons donc travailler en nous limitant à l'utilisation de techniques tout à fait classiques de programmation, et dans le cas présent notre énumération va se traduire par une simple instruction **If** ... **EndIf** assortie de plusieurs cas.

La programmation de notre fonction ne sera cependant pas totalement naïve.

Comme le montre l'énumération précédente, la dérivation d'une expression fait intervenir le résultat obtenu en dérivant une ou plusieurs sous-expressions de cette expression.

Notre fonction de dérivation va donc être récursive et s'appeler elle-même.

Naturellement ce processus s'arrête de lui-même lorsque l'on arrive à une sous-expression suffisamment simple, c'est à dire à une variable ou à une constante (cf. premier cas.).

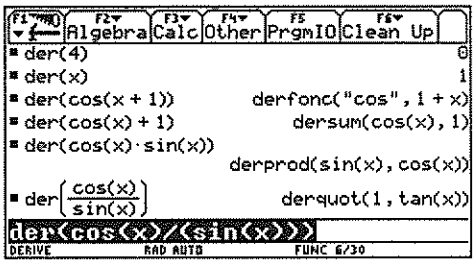
Pour alléger un peu l'écriture de nos différentes fonctions, nous allons supposer que la fonction que nous souhaitons dériver est une fonction de x.

Nous allons à présent pouvoir écrire la fonction de dérivation :

```
der(ex)
Func
Local  ex1,ex2,op,fonc
If part(ex)=Ø Then
  when(ex=x,1,Ø,Ø)
ElseIf part(ex)=1 Then
  part(ex,Ø)→fonc
  part(ex,1)→ex1
  derfonc(fonc,ex1)
ElseIf part(ex)=2 Then
  part(ex,Ø)→op
  part(ex,1)→ex1
  part(ex,2)→ex2
  If op="+" Then
    dersum(ex1,ex2)
  ElseIf op="-" Then
    derdif(ex1,ex2)
  ElseIf op="*" Then
    derprod(ex1,ex2)
  ElseIf op="/" Then
    derquot(ex1,ex2)
  ElseIf op="^" Then
    derpuiss(ex1,ex2)
Else
  d(ex,x)
EndIf
EndFunc
```

Comme on peut le voir, cette fonction fait appel à d'autres fonctions : **derfonc**, **dersum**, **derdif**, **derprod**, **derquot**, **derpuiss** qui seront chargées de traiter les cas des expressions du type $f(u(x))$, les sommes, différences, produits, quotients et puissances d'expressions. Ces fonctions n'existent pas encore. Nous les définirons ultérieurement.

Voici ce que donnent quelques essais de notre fonction, dans l'état actuel d'avancement du projet :



On peut facilement voir que la fonction **der** effectue bien le tri en fonction des différents cas, et qu'elle appelle la fonction de dérivation adaptée à chaque situation, en transmettant les arguments dont on a besoin pour poursuivre les calculs.

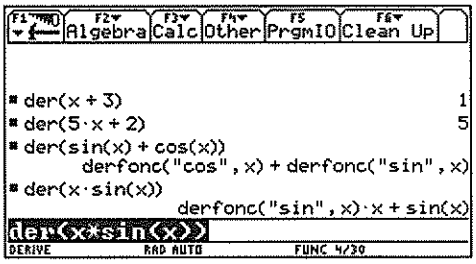
Naturellement, le calcul s'arrête aussitôt puisque les différentes fonctions n'ont pas encore été définies.

Définissons par exemple les fonctions **dersum** et **derprod** :

```
dersum(ex1,ex2)
Func
der(ex1)+der(ex2)
EndFunc

derprod(ex1,ex2)
Func
der(ex1)*ex2+ex1*der(ex2)
EndFunc
```

Nous pouvons à présent faire quelques essais supplémentaires :



Nous allons maintenant aller un peu plus loin en définissant la fonction **derfonc** qui est chargée de dériver les expressions du type $f(u(x))$.

La bonne méthode consiste à gérer une table dans laquelle sont placés les noms de fonctions et la formule à utiliser pour les dériver.

Pour éviter de trop entrer dans des détails techniques, nous allons nous contenter d'utiliser une fonction qui sera capable de dériver les fonctions trigonométriques et logarithmes. C'est essentiellement cette partie qu'il faudrait développer pour obtenir un noyau de dérivation vraiment performant.

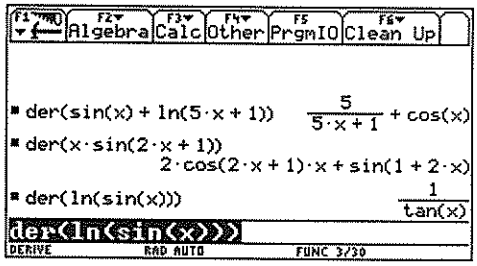
Pour écrire cette fonction, il faut également savoir qu'une expression comme $-\cos(x)$ est vue par la TI-89 ou la Voyage 200 comme le résultat obtenu en appliquant la fonction "opposé" à l'expression $\cos(x)$. Dans la pratique, la valeur retournée par **part**(ex,0) sera alors égale à `"-"`.

Cela correspond au premier cas étudié dans le texte de la fonction : `If op="-" Then...`

Les cas suivants correspondent à l'utilisation des fonctions cosinus, sinus, tangente et logarithme népérien. Il est en revanche inutile de prévoir le cas de la fonction exponentielle car celui-ci est traité par la fonction chargée de dériver les puissances.

```
derfonc(op,arg)
Func
Local  a
der(arg)→a
If op="-" Then
 -a
ElseIf op="cos" Then
 -a*sin(arg)
ElseIf op="sin" Then
 a*cos(arg)
ElseIf op="tan" Then
 a*(1+(tan(arg))^2)
ElseIf op="ln" Then
 a/arg
EndIf
EndFunc
```

Quelques nouveaux essais permettent de se rendre compte des progrès réalisés par notre système de dérivation avec l'ajout des dernières fonctions :



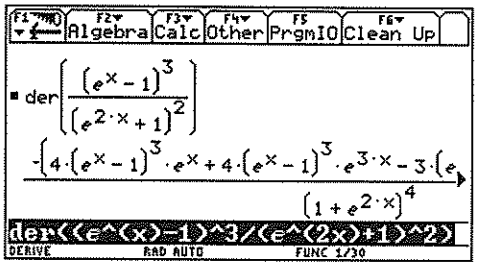
Il ne reste plus qu'à écrire les dernières fonctions permettant de dériver une différence, un quotient ou une puissance pour aller encore plus loin.

```
derdif(ex1,ex2)
Func
der(ex1)-der(ex2)
EndFunc

derquot(ex1,ex2)
Func
(der(ex1)*ex2-ex1*der(ex2))/ex2^2
EndFunc

derpuiss(ex1,ex2)
Func
(der(ex1)*ex2/ex1+ln(ex1)*der(ex2))*ex1^ex2
EndFunc
```

À présent, notre module de dérivation permet d'effectuer des calculs qui commencent à devenir un peu compliqués :



En bref, il y a effectivement de quoi se montrer prudent lors de l'utilisation de ces outils.

Mais il faut rappeler que certains formulaires, donnant la liste des primitives de fonctions "classiques", et que l'on utilisait précédemment, comportaient jusqu'à 10% d'erreurs !

Il serait donc totalement absurde de rejeter les logiciels de calcul formel sous ce prétexte.

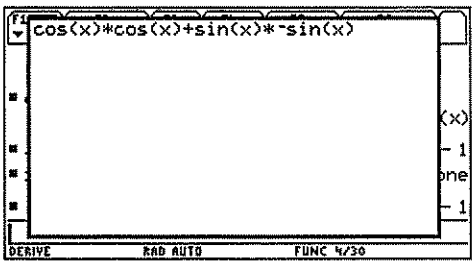
Cela montre en revanche combien il serait illusoire de penser pouvoir manipuler de tels outils sans disposer des connaissances mathématiques indispensables pour savoir en déjouer les pièges et interpréter correctement les résultats fournis.

Les logiciels de calcul formel sont de fantastiques outils.

Ils ne remplacent en aucun cas une étude approfondie des Mathématiques !

Tout cela peut sembler un peu mystérieux...

- Avec la première méthode, c'est l'utilisation de la variable *a* qui provoque une évaluation/simplification de son contenu. Cette variable contient en fait l'expression non simplifiée.
  On peut le vérifier en utilisant la touche **F6:Contents** dans le menu **VAR-LINK**, qui permet de visualiser le contenu d'une variable, tel qu'il est sans aucune modification.
  Voici ce que l'on obtient :



- Dans le second cas, lors de l'utilisation de la fonction **ld**, l'argument qui est égal à la valeur (non simplifiée) de l'expression va être évalué et simplifié avant d'être transmis à cette fonction.
  C'est donc lors de l'appel de cette fonction que la simplification a lieu.
  Ensuite, la fonction se contente de renvoyer la valeur telle qu'elle l'a reçue.

Tout cela permet d'entrevoir la complexité des mécanismes d'évaluation et de simplification mis en jeu dans les logiciels de calculs symboliques.

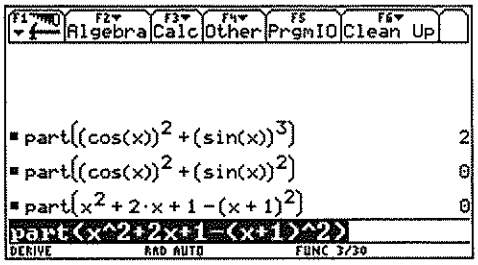
Le choix de la façon dont ces règles seront utilisées est spécifique à chaque système de calcul formel.

Dans le cas de la TI-89 / Voyage 200, une simplification importante est effectuée dès que l'on utilise une expression, comme par exemple lors de l'appel d'une fonction.

La majorité des utilisateurs peuvent éventuellement ne pas être au courant de ce type de détail.

Par contre, cela peut prendre une importance capitale si l'on cherche à programmer du calcul formel.

Par exemple, pour comprendre les résultats fournis par la fonction **part** dans l'écran ci-dessous (et donc ne pas faire d'erreur lors de son utilisation), il est indispensable de savoir qu'une simplification a lieu lors de l'appel de cette fonction.



Dans le premier cas, on est resté avec la même expression, car celle-ci n'a pas pu être simplifiée par la calculatrice. C'est donc bien une somme de deux termes qui a été analysée par **part**.

Par contre, dans les deux cas suivants, la simplification préalable a conduit à l'analyse d'expressions constantes. Tout se passe comme si l'on avait en fait calculé **part**(1) et **part**(0).

C'est un choix des concepteurs du logiciel présent sur cette calculatrice, mais il n'y a pas de règle uniforme en la matière.

Par exemple, avec Maple on dispose de la fonction **op** pour récupérer la liste des sous-expressions. Lors d'un appel de la forme `op(3^2+2*3+1-(3+1)^2)`, l'expression qui ne fait intervenir que des entiers sera automatiquement simplifiée, mais ce ne sera pas le cas avec `op(x^2+2*x+1-(x+1)^2)`.

Disposer d'un bon outil ne dispense pas de réfléchir !

Un des principaux problèmes que l'on rencontre lors de l'utilisation de systèmes de calcul formel par des étudiants néophytes dans ce domaine est l'absence, parfois totale, de réflexion avant de se lancer dans des calculs pourtant voués à l'échec.

Il suffit pourtant d'exemples très simples pour voir comment un minimum de travail préalable peut parfois considérablement changer la nature des calculs à accomplir.

Considérons par exemple la suite définie par

$$\begin{cases} u_0 = 1 \\ u_n = u_{n-1}^2 + 2u_{n-1} \end{cases}$$

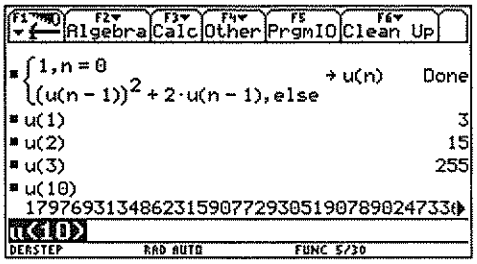
On dispose sur la TI-89, la TI-92 Plus et la Voyage 200 d'une manière particulièrement simple de programmer une suite de ce type.

Il suffit d'utiliser une instruction du type

```
when(n=0,1, u(n-1)^2+2* u(n-1)) → u(n)
```

pour définir la fonction u permettant de calculer les termes par récurrence.

Voyons ce que l'on obtient dans l'écran ci-dessous :



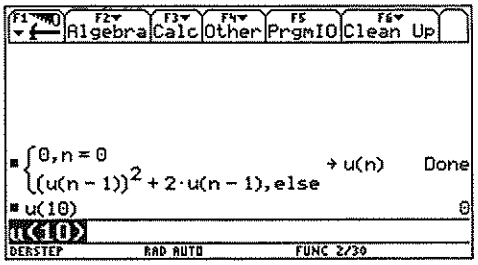
Apparemment, ça marche…

Cependant, le calcul de u_{10} a nécessité près de 55 secondes…

Si vous avez déjà fait de la programmation récursive, vous connaissez sans doute l'origine du problème… mais adoptons ici le point de vue d'un utilisateur débutant dans ce domaine.

Une première "explication" serait de penser que la calculatrice a tout simplement eu du mal à effectuer tous les calculs exacts sur les grands entiers intervenant ici. Ce n'est pas totalement faux, et on peut en effet se demander s'il est bien judicieux d'effectuer un calcul exact dans ce cas précis.

Mais ce n'est pas la seule raison du problème, loin de là.
Pour s'en convaincre, il suffit de choisir la valeur $u_0 = 0$, qui conduit à $u_n = 0$ pour toutes les valeurs de n. Cette fois les calculs ne devraient pas présenter de difficultés…



Il a encore fallu environ 50 secondes pour obtenir ce dernier résultat, le calcul du terme suivant u_{11} nécessite près d'une 1'40"…
Et pourtant, on ne fait que manipuler des nombres qui sont tous égaux à 0…

Il faut donc chercher une autre explication, qui va s'appuyer sur le nombre de fois où la fonction **u** va devoir être utilisée lors de notre calcul de u_n.

Le nombre a_n d'appels nécessaires à la fonction **u** pour calculer u_n est de

- 1 pour calculer u_0
- 3 pour calculer $u_1 = u_0^2 + 2u_0$: on doit calculer $u(1)$ – appel n°1 –, ce qui va automatiquement lancer le calcul de $u(0)$ – appel n°2 – que l'on va élever au carré, suivi d'un *nouveau* calcul de $u(0)$ – appel n°3 – qui sera multiplié par 2.

De même, pour les valeurs supérieures, le calcul de u_{n+1} va utiliser deux calculs de u_n, et le nombre d'appels a_n vérifie donc: $a_n = 1 + 2a_{n-1}$. (On trouve ici l'explication du doublement du temps de calcul nécessaire lorsque l'on passe de u_{10} à u_{11}).

Pour calculer a_n, il suffit de remarquer que la suite $w_n = a_n + 1$ est géométrique, de premier terme $w_0 = 1 + 1 = 2$, et de raison 2. On en déduit $w_n = 2^n \cdot 2 = 2^{n+1}$ et donc $a_n = 2^{n+1} - 1$

En fait, une simple transformation de la formule de récurrence permet ici de faire disparaître totalement ce problème. Il suffit par exemple d'utiliser une mise sous forme canonique :

$$\begin{cases} u_0 = 1 \\ u_n = (1 + u_{n-1})^2 - 1 \end{cases}$$

Ce qui peut se traduire par

```
when(n=0,1, (1+u(n-1))^2-1) →u(n)
```

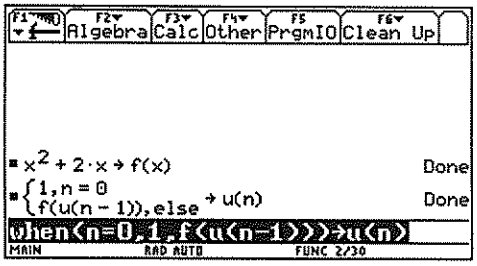
Le calcul est alors beaucoup plus rapide, et seule l'opération d'affichage du résultat nécessite un peu de temps.

Cela s'explique facilement. Le nombre d'appels nécessaires vérifie à présent la relation de récurrence

$$\begin{cases} a_0 = 1 \\ a_n = a_{n-1} + 1 \end{cases}$$

Et on a donc $a_n = n + 1$.

Une autre solution aurait consisté à utiliser une fonction intermédiaire pour définir le calcul à effectuer par récurrence, comme on le fait dans l'écran ci-dessous :



En effet, ici aussi, on ne fait appel qu'une seule fois au calcul de u_{n-1} dans le calcul de u_n. Le résultat est ensuite passé en argument à la fonction *f*, qui se charge d'effectuer le calcul nécessaire à partir de la valeur transmise. Le nombre d'appels de la fonction **u** est donc ici aussi égal à $a_n = n + 1$.

Attention cependant, cela ne suffira pas à résoudre les éventuels problèmes liés à la taille des nombres manipulés. On a ici $u_{10} \approx 1{,}80 \times 10^{308}$ et $u_{11} \approx 3{,}23 \times 10^{616}$.

Le calcul de u_{12} provoquera immanquablement un dépassement des capacités de la calculatrice…