Au fond, c'est quoi un langage de programmation ?

Yann Orlarey

JIM 2015
Montréal 7–9 mai 2015

Langages musicaux

Exemples de langages et de formalismes utilisés en musique

- 4CED
- abc
- Adagio
- AML
- AMPLE
- Arctic
- Autoklang
- Bang
- Canon
- CHANT
- **Chuck**
- CLCE
- CMIX
- Cmusic
- CMUSIC
- Common Lisp Music
- Common Music
- Common Music Notation
- **Csound**
- CyberBand
- DARMS
- DCMP
- DMIX
- **Elody**
- EsAC
- Euterpea
- Extempore
- **Faust**
- Flavors Band
- Fluxus
- FOIL
- FORMES
- FORMULA
- Fugue
- Gibber
- GROOVE
- GUIDO
- HARP
- Haskore
- HMSL
- INV
- invokator
- KERN
- Keynote
- Kronos
- Kyma
- LOCO
- LPC
- Mars
- MASC
- Max
- MidiLisp
- MidiLogo
- MODE
- MOM
- Moxc
- MSX
- MUS10
- MUS8
- MUSCMP
- MuseData
- MusES
- MUSIC 21
- MUSIC 10
- MUSIC 11
- MUSIC 360
- MUSIC 4B
- MUSIC 4BF
- MUSIC 4F
- MUSIC V
- MUSIC 6
- MCL
- **MUSIC III/IV/V**
- MusicLogo
- Music1000
- MUSIC7
- Musictex
- MUSIGOL
- MusicXML
- Musixtex
- NIFF
- NOTELIST
- Nyquist
- OPAL
- OpenMusic
- Organum1
- Outperform
- Overtone
- PE
- Patchwork
- PILE
- Pla
- PLACOMP
- PLAY1
- PLAY2
- PMX
- POCO
- POD6
- POD7
- PROD
- **Puredata**
- PWGL
- Pyo
- Ravel
- SALIERI
- SCORE
- ScoreFile
- SCRIPT
- SIREN
- SMDL
- SMOKE
- SSP
- SSSP
- ST
- **Supercollider**
- Symbolic Composer
- Tidal

Au fond, c'est quoi un langage de programmation ?

Pour répondre à cette question, l'idée c'est d'enlever tout ce qui est superflu et de voir ce qui reste, les notions vraiment primitives. Par exemple : est-ce que on a besoin des nombres pour faire un langage de programmation ?

Pour cela on va voir trois formalismes de radicalité croissante :

- Les systèmes de réécriture
- Le lambda-calcul
- La logique combinatoire

Systèmes de Réécriture

Système de réécriture

Un *système de réécriture* (Axel Thue, 1914) est une façon de décrire des transformations à partir de simples règles de remplacement, appelées *règles de réécriture*.

Exemple :

On a une situation de départ, ici une série de grains de café noir ou blanc :

blanc blanc noir noir blanc blanc noir noir

Et les règles de réécriture suivantes :

$$\begin{aligned} \text{noir blanc} &\rightarrow \text{noir} \quad (r1) \\ \text{blanc noir} &\rightarrow \text{noir} \quad (r2) \\ \text{noir noir} &\rightarrow \text{blanc} \quad (r3) \end{aligned}$$

Système de réécriture

Partant de la situation de départ, on va appliquer les règles de réécriture jusqu'à arriver à une situation terminale où plus aucune règle ne s'applique. Cette situation terminale est appelée en **forme normale**.

Exemple de transformation

blanc blanc noir noir blanc blanc noir noir	(r1)
blanc blanc noir **noir** blanc noir noir	(r3)
blanc blanc **blanc** blanc noir noir	(r2)
blanc blanc blanc **noir** noir	(r2)
blanc blanc **noir** noir	(r2)
blanc **noir** noir	(r2)
noir noir	(r3)
blanc	forme normale

Système de réécriture

Dans notre exemple, à partir d'une même situation de départ on peut arriver à une forme normale différente suivant le choix d'application des règles de réécritures.

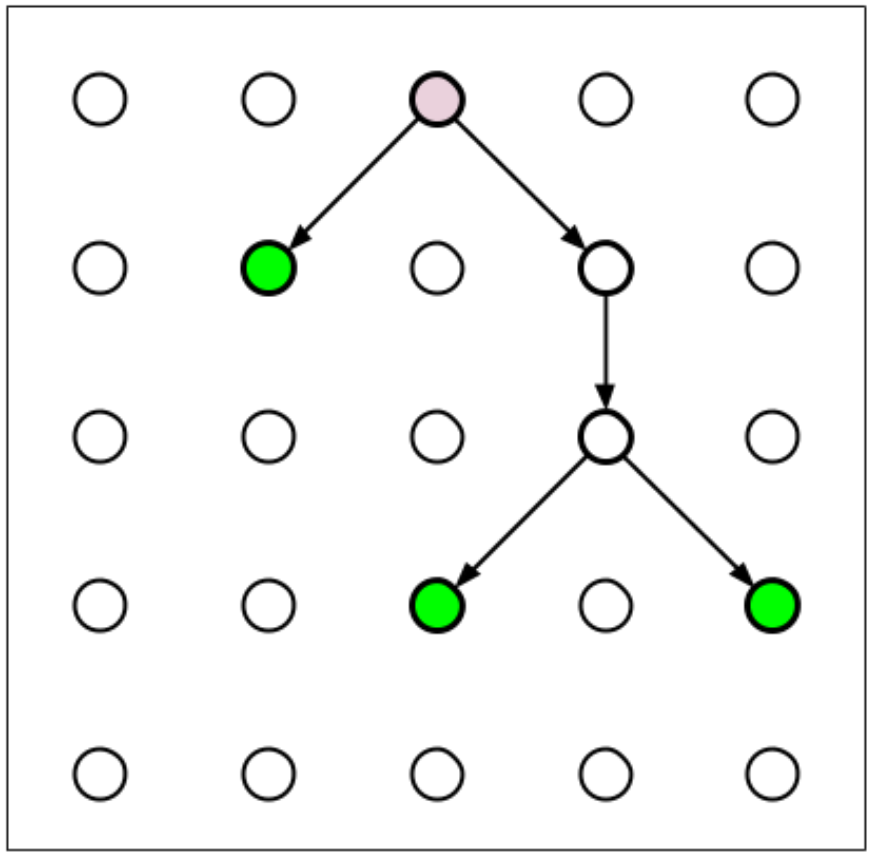
Deuxième exemple de transformation

blanc blanc noir noir blanc blanc noir noir	(r3)
blanc blanc **blanc** blanc blanc noir noir	(r3)
blanc blanc blanc blanc blanc **blanc**	forme normale

Quand on peut, à partir d'une même situation de départ, arriver à des formes normales différentes, on dit que *le système de réécriture n'est pas confluent*.

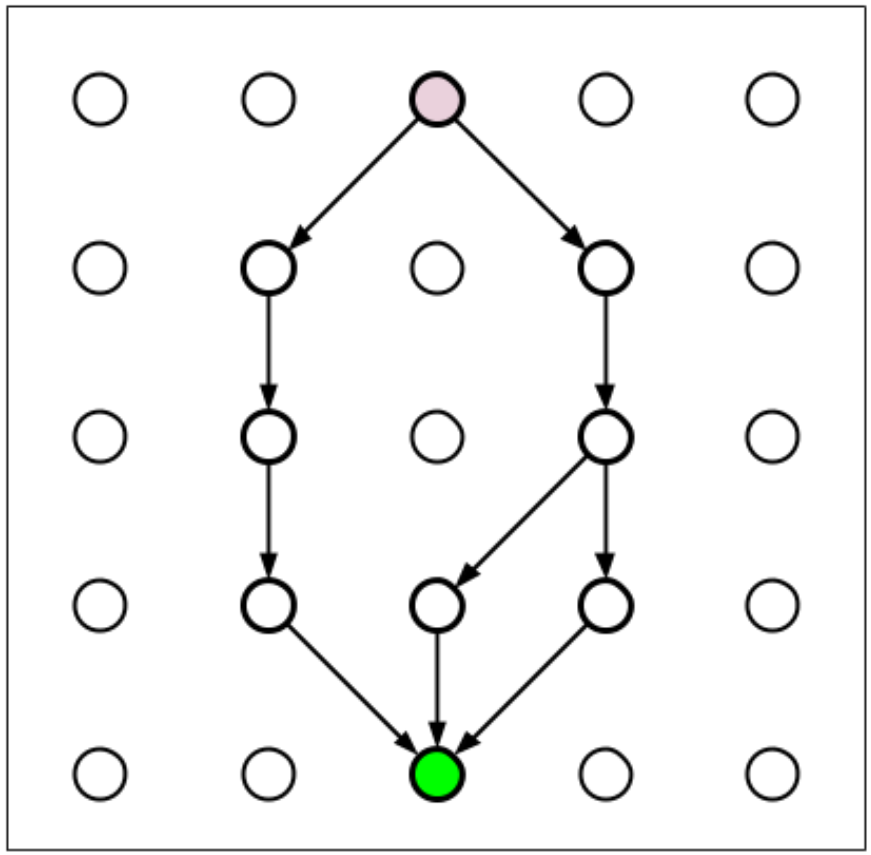
Système de réécriture

Système non-confluent. A partir d'un même point de départ plusieurs formes normales différentes peuvent être atteintes. L'ordre d'application des règles conditionne le résultat.



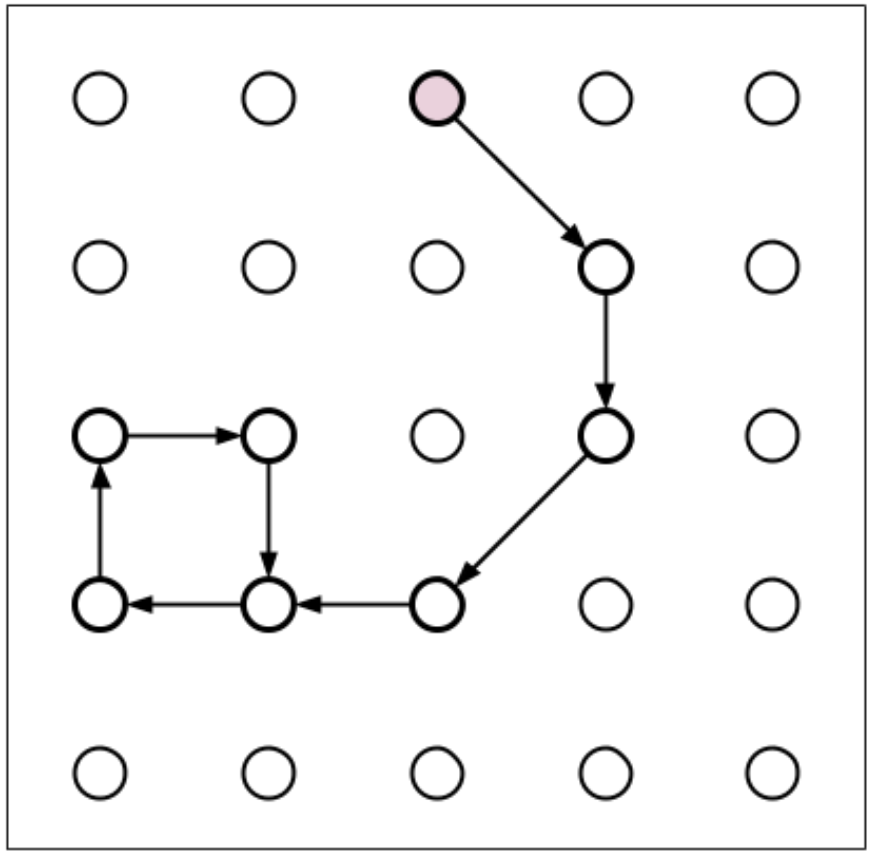
Système de réécriture

Système confluent. Un système de réécriture est *confluent* si l'on abouti toujours à la même forme normale quelque soit l'ordre d'application des règles de réécriture. Il représente un *calcul* et la forme normale est la *valeur* de l'expression de départ.



Système de réécriture

Certains systèmes de terminent pas, on n'atteint jamais une forme normale.



Système de réécriture

Les systèmes de réécriture sont par exemple utilisés en musique.

Pachet, François, Sur la structure algébrique des séquences d'accord, Journées d'Informatique Musicale, Agelonde, May, 1998.

Sur la structure algébrique des séquences d'accords de Jazz

François Pachet
SONY CSL, 6 rue Amyot, 75005 Paris
Email: pachet@csl.sony.fr

Résumé
Dans le cadre du projet d'*écoute active* mené à Sony CSL, nous nous intéressons à la notion de règle de substitution, permettant de faire des variations de structures harmoniques de séquences d'accords de Jazz. Nous décrivons un système permettant de mettre ces règles en oeuvre de manière intuitive pour produire des séquences d'accords "intéressantes". Nous proposons une représentation récursive des règles qui permet à l'utilisateur de fabriquer ses propres règles. Enfin, nous soulevons quelques problèmes théoriques non résolus.

1. De la forme ouverte à l'écoute active

Le travail présenté ici s'inscrit dans le contexte du projet "Ecoute Active" développé au laboratoire Sony-CSL. Ce projet consiste à chercher des moyens d'enrichir la pratique de l'écoute musicale d'oeuvres existantes, en donnant à l'auditeur des moyens de modifier son écoute selon certains critères. Ces critères ou paramètres définissent des espaces dans lesquels un auditeur peut se déplacer pendant l'écoute d'un morceau de musique, en faisant varier certaines de ses caractéristiques. Les points importants de notre contexte de travail sont 1) l'exploitation de pièces musicales existantes, préenregistrées, et 2) l'identification de variations sémantiquement cohérentes sur ces pièces musicales. Ce projet d'écoute active se distingue ainsi d'autres efforts de recherche en musique et interaction: D'une part, par opposition à la notion de "forme ouverte" (Eckel, 1997), nous nous intéressons à construire des environnements dans lesquels les variations musicales préservent le sens original, pour autant que l'on puisse le définir avec précision. Par ailleurs, nous ne cherchons pas à décomposer un signal numérique musical en différentes composantes perceptives, comme proposé par Lepain (1998) dans le cadre de la consultation interactive de documents musicaux, mais tentons plutôt d'extraire des possibilités de variation d'un document donné qui "font sens" musicalement.

Dans ce contexte, nous nous sommes intéressé à une dimension de variation liée à la structure harmonique de certaines pièces de Jazz. En effet, il se trouve que pour un corpus assez large de morceaux dits "standards", tels que ceux trouvés dans le Real Book (Real, 1981) ou le Fake Book (Fake, 1983), on peut faire apparaître une structure harmonique abstraite qui comporte certaines propriétés algébriques rendant possible de telles variations, au sens ou 1) des séquences existantes peuvent être modifiées selon des règles précises et 2) la sémantique d'une séquence peut être préservée, dans un certain sens, par ces modifications. Ce papier décrit ces structures algébriques de manière informelle, et propose un outil expérimental permettant de produire des variations autour de grilles d'accords.
Dans la section 2, nous introduisons la notion de règle de substitution de Jazz, en la décrivant comme une règle de réécriture, et mettons en évidence son caractère combinatoire. Dans les sections 3 et 4 nous proposons une représentation des accords et des grilles d'accord servant de support à la mise en oeuvre de ces règles. Dans la section 5 nous proposons une base de règles permettant d'engendrer des variations intéressantes à partir de grilles de base, et montrons comment implémenter un mécanisme simple pour mettre ces règles en oeuvre. Enfin nous montrons comment la représentation réflexive des règles de substitution permet de créer de nouvelles règles, elles-mêmes "licites". Nous concluons en soulevant quelques problèmes théoriques.

L'article "*Sur la structure algébrique des séquences d'accords de Jazz*" (Pachet, JIM 1998) propose plusieurs règles de réécriture pour le Jazz :

$$
\begin{aligned}
C7 &\rightarrow F\#7 && \text{(subst. au triton)} \\
C &\rightarrow G7 \,/\, C && \text{(prép. par septième)} \\
G7 &\rightarrow Dmin7 \,/\, G7 && \text{(prép. par min7)}
\end{aligned}
$$

Système de réécriture

Les systèmes de réécriture sont par exemple utilisés en musique.

A Term Rewriting based Structural Theory of Rhythm Notation
Research report – ANR-13-JS02-0004-01

Florent Jacquemard[1], Pierre Donat-Bouillud[1,2], and Jean Bresson[1]

[1] UMR STMS: IRCAM-CNRS-UPMC and INRIA, Paris, France.
[2] ENS Rennes, Ker Lann Campus, France.

This report is the extended version of a paper presented at the *International Conference on Mathematics and Computation in Music – MCM 2015*).

Abstract. We present a tree-based symbolic representation of rhythm notation suitable for processing with purely syntactic theoretical tools such as term rewriting systems or tree automata. Then we propose an equational theory, defined as a set of rewrite rules for transforming these representations. This theory is complete in the sense that from a given rhythm notation the rules permit to generate all notations of equivalent durations. It can be used to explore the space of Using complementary tree automata formalisms, one can restrict the search space according to notations preferences regarding e.g. metre or other user defined syntactic constraints.

Introduction

Term Rewriting Systems (TRSs) [8] are well established formalisms for tree processing (transformation and reasoning). With solid theoretical foundations, they are used in a wide range of applications, to name a few: automated reasoning, natural language processing, foundations of Web data, etc. TRSs perform in-place transformations in trees by the replacement of patterns, as defined by oriented equations called rewrite rules. They are a classical model for symbolic computation, used for rule-based modeling, simulation and verification of complex systems or software (see e.g. the languages TOM[3] and Maude[4]). Tree Automata (TAs) [7] are finite state recognizers of trees which permit to characterize specific *types* of tree-structured data (*regular* tree languages). They are often used in conjunction with TRSs, acting as *filters* in the explorations of sets of trees computed by rewriting.

It is also common to use trees to represent hierarchical structures in symbolic music (see [15] for a survey). For instance, the GTTM [13] uses trees to analyse inner relations in musical pieces. Trees are also a natural representation of rhythms, where durations are expressed as a hierarchy of subdivisions. Computer-aided composition (CAC) environments such as Patchwork and OpenMusic [3,6] use structures called *rhythm trees* (RTs) for representing and programming rhythms [2]. Such hierarchical, notation-oriented approach (see also [15]) is complementary to the performance-oriented formats corresponding to the MIDI notes' onsets and offsets in standard computer music systems. It also provides a more structured representation of time than music notation formats such as MusicXML [9] or Guido [12], where durations are expressed with integer values. As highly structured representations, trees enable powerful manipulation and generation processes in the rhythmic domain (see for instance [11]), and enforce some structural constraints on duration sequences.

In this paper, we propose a tree-structured representation of rhythm suitable for defining a set of rewriting rules (*i.e.* oriented equations) preserving rhythms, while allowing simplifications of notation. This representation bridges CAC rhythm structures with formal tree-processing approaches, and enables a number of new manipulations and applications in both domains. In particular, rewriting rules can be seen as an axiomatization of rhythm notation, which can be applied to reasoning on equivalent notations in computer-aided music composition or analysis.

After some preliminary definitions in Section 1, we introduce our tree representations of rhythms in Section 2. Section 3 presents the rewriting rules proposed based on this representation, and Section 4 finally states some properties of this general representation framework.

[3] http://tom.loria.fr
[4] http://maude.cs.illinois.edu

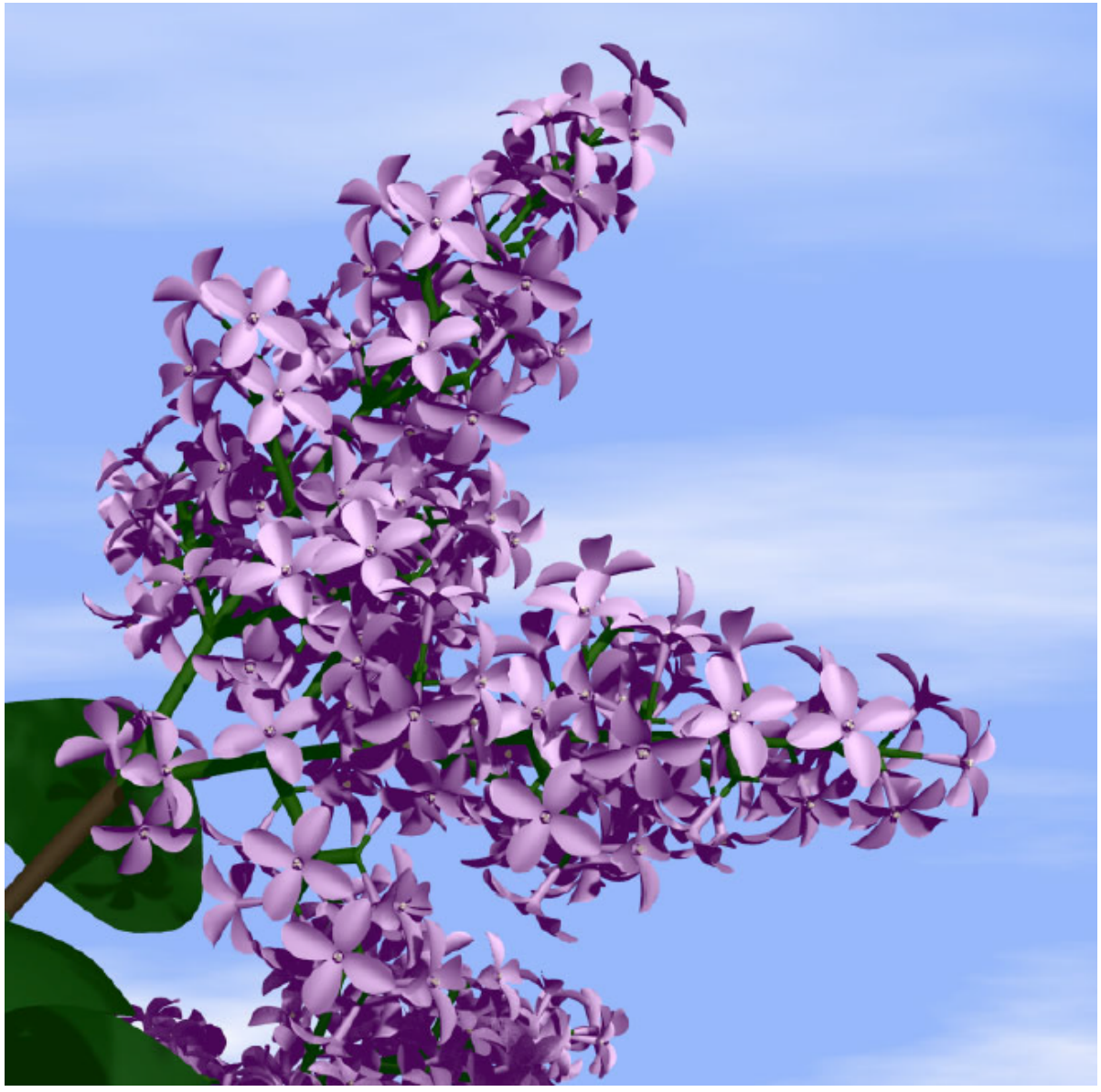
L'article "*A Term Rewriting based Structural Theory of Rhythm Notation*" (Florent Jacquemard, Pierre Donat-Bouillud et Jean Bresson) propose l'utilisation de règles de réécriture pour la notation de rythmes complexes.

Système de réécriture

Une utilisation spectaculaire des systèmes de réécriture (dans une variante appelée *système de Lyndenmayer* ou *L-system*) est la modélisation de la croissance des plantes.

L'un des meilleurs spécialistes se trouve au Canada : Przemyslaw Prusinkiewicz. Il dirige le *Biological Modeling and Visualization research group* à l'université de Calgary `http://algorithmicbotany.org`.

Système de réécriture



Système de réécriture

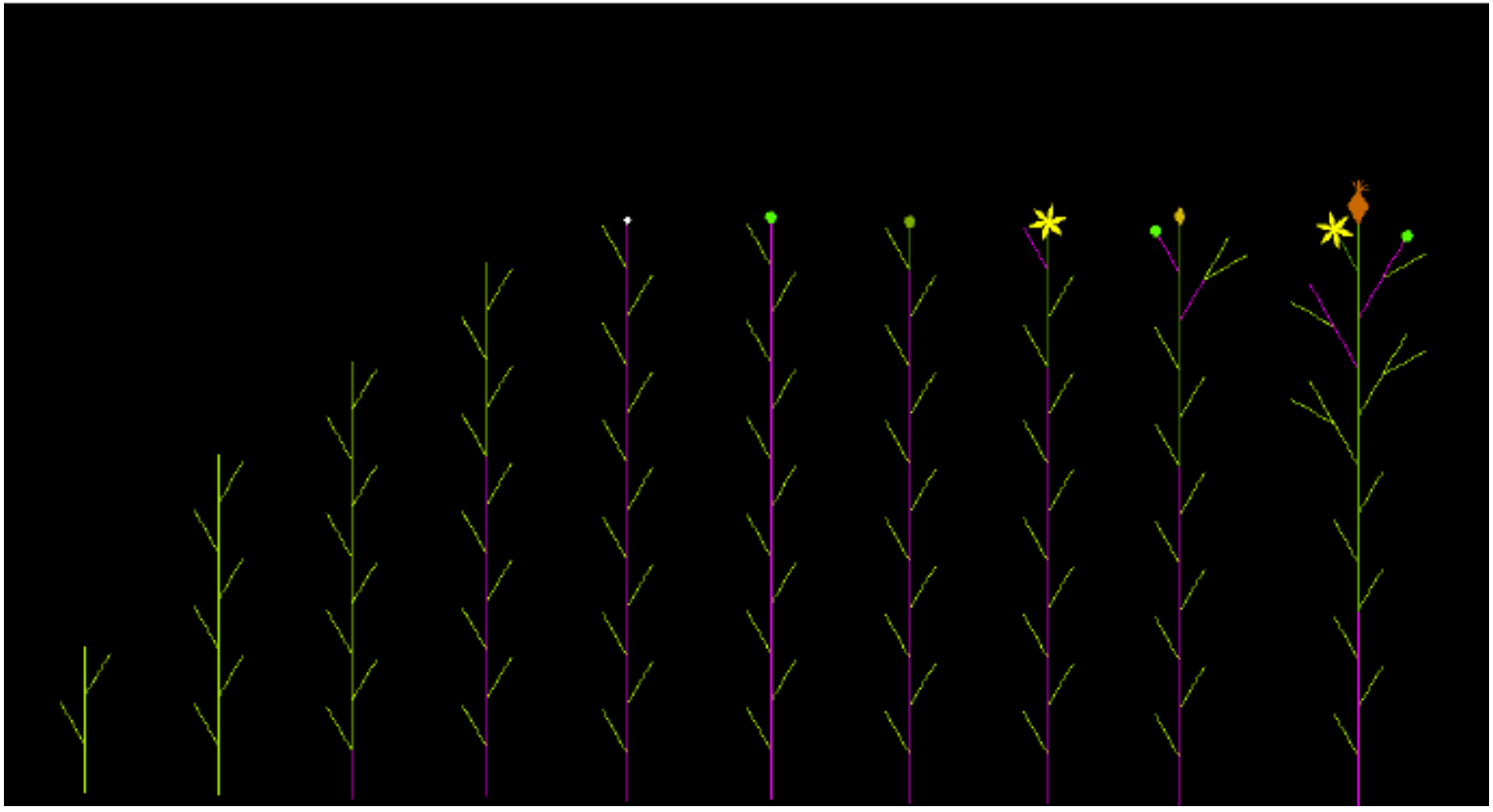
Modélisation de Mycelis muralis : le code.

```
#include K                    /* flower shape specification */
#consider M S T V

ω  :  I(20)FA(0)
p1 :  S < A(t)  :*     → TV K
p2 :  V < A(t)  :*     → TV K
p3 :      A(t)  :t>0   → A(t-1)
p4 :      A(t)  :t=0   → M[+(30)G]F/(180)A(2)
p5 :  S < M     :*     → S
p6 :      S > T :*     → T
p7 :  T < G     :*     → FA(2)
p8 :  V < M     :*     → S
p9 :      T > V :*     → W
p10:      W     :*     → V
p11:      I(t)  :t>0   → I(t-1)
p12:      I(t)  :t=0   → S
```

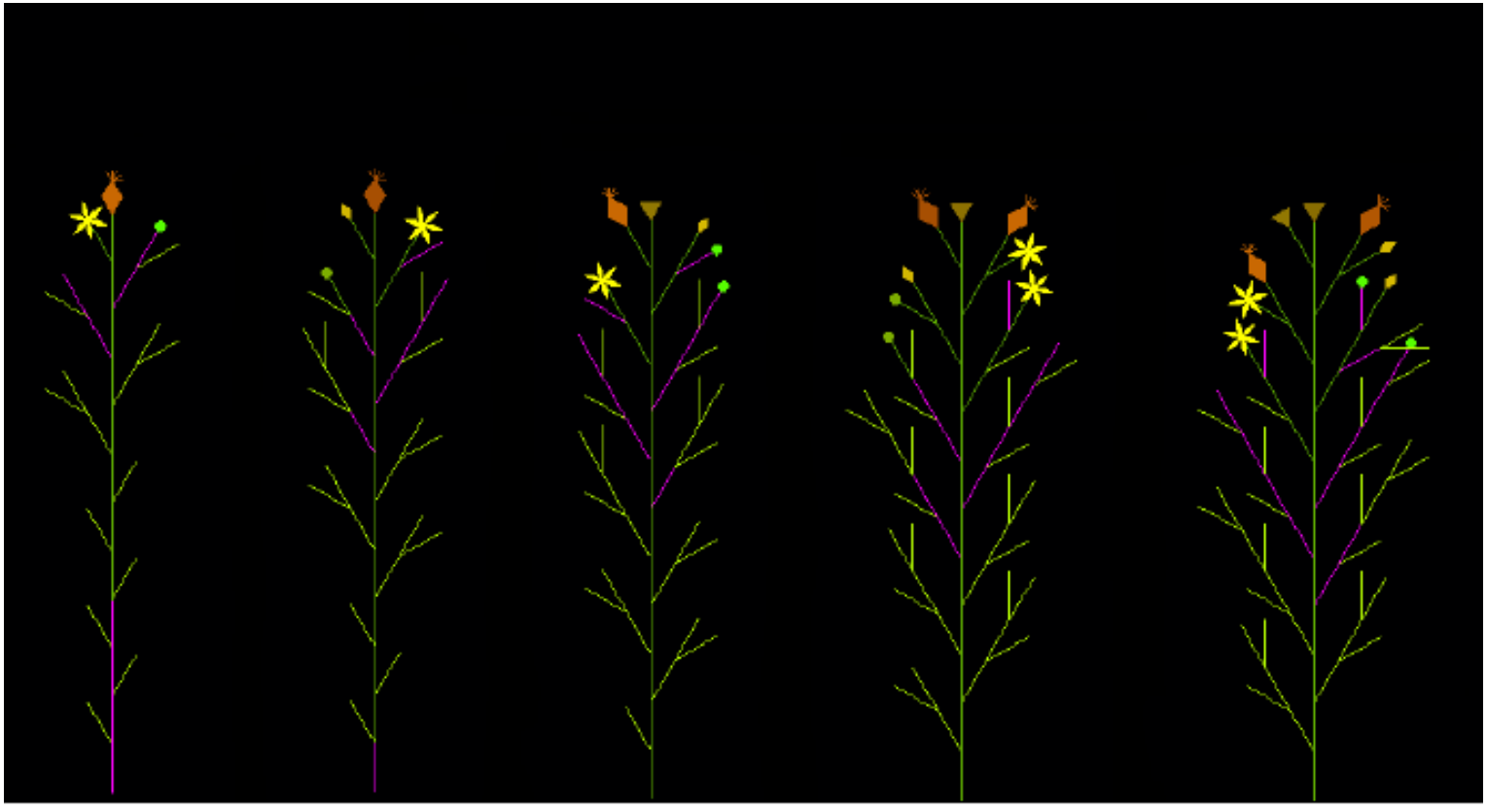
Système de réécriture

Modélisation de Mycelis muralis : premières itérations.



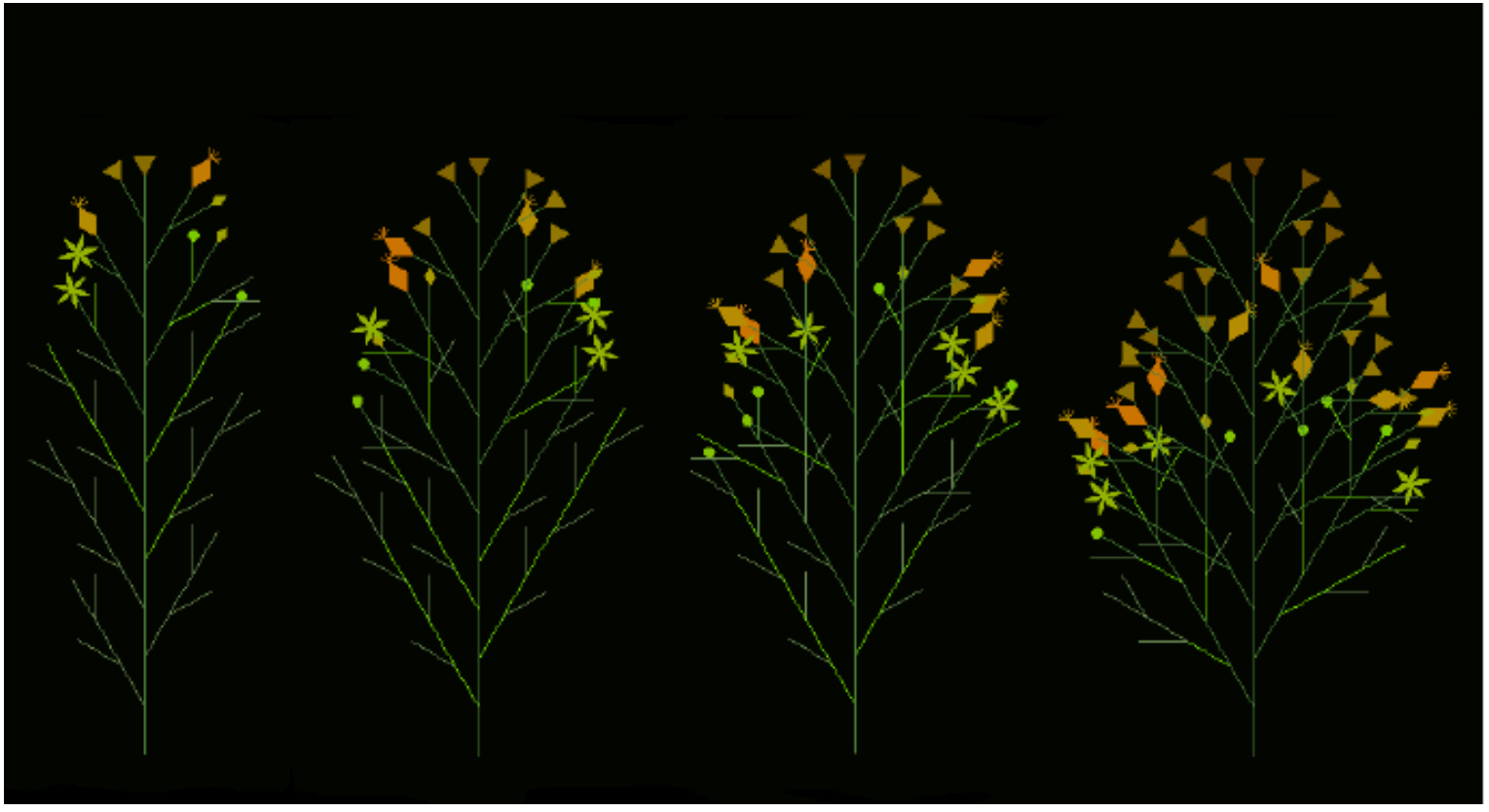
Système de réécriture

Modélisation de Mycelis muralis : itérations suivantes.



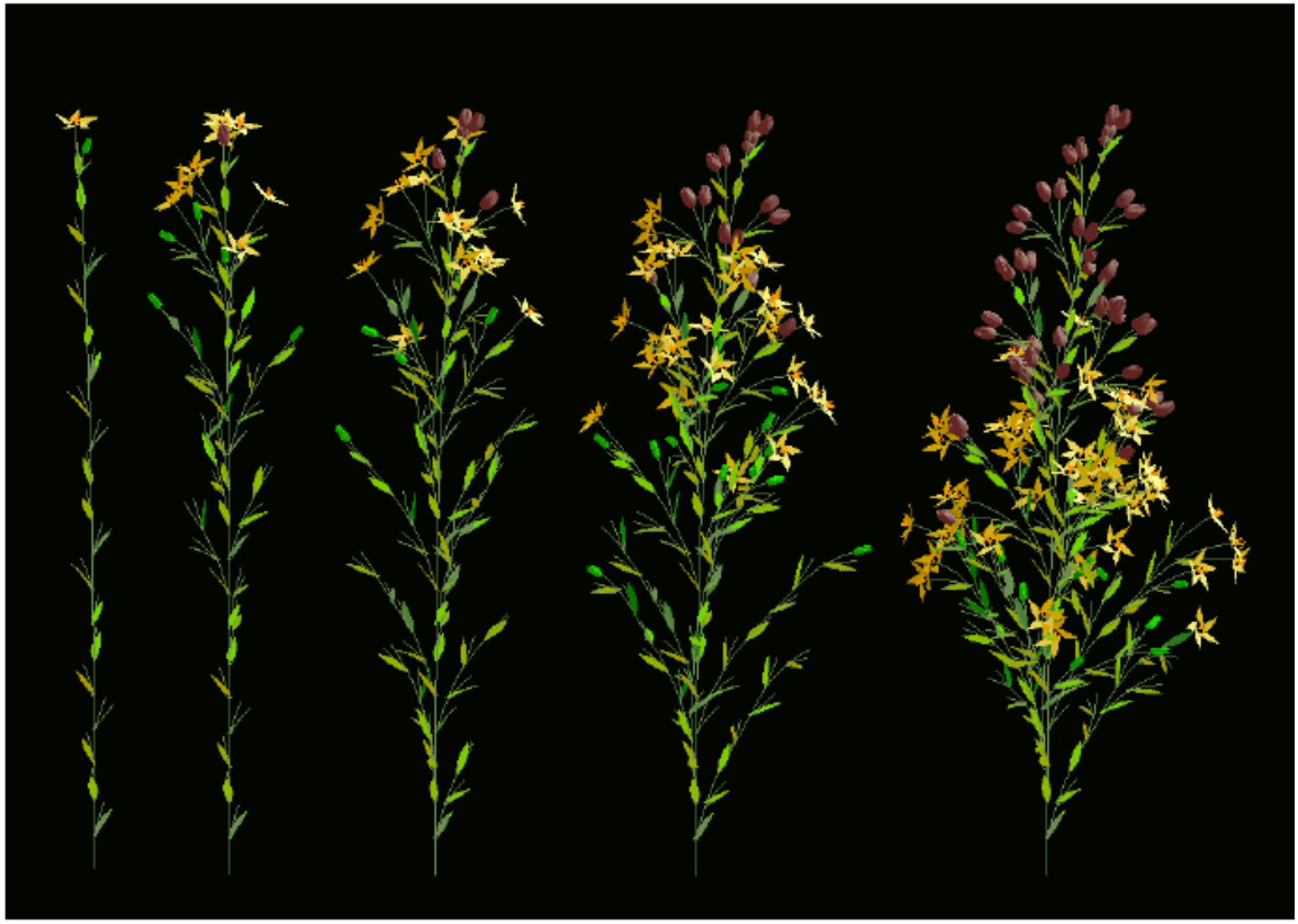
Système de réécriture

Modélisation de Mycelis muralis : itérations finales.



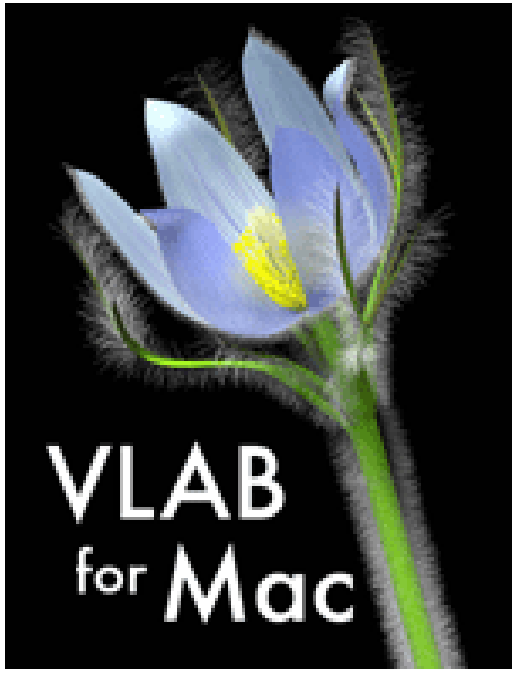
Système de réécriture

Modélisation de Mycelis muralis : modèle 3D.



Système de réécriture

Démonstration VLAB for Mac.



Lambda-Calcul

Lambda Calcul

- Le lambda-calcul a été inventé à Princeton par Alonzo Church dans les années 30.
- C'est au départ une théorie qui vise à re-fonder les mathématiques sur la notion de fonction plutôt que d'ensemble. Mais cette tentative échoue pour cause de paradoxes.

- Le lambda-calcul que l'on connait aujourd'hui est une théorie axiomatique de la notion de fonction que l'on peut voir comme un **langage de programmation fonctionnel réduit à son essence**.
- Alonzo Church "The calculi of lambda conversion", Princeton University Press, 1941.

Lambda Calcul

Le Lambda-Calcul (λ-Calcul) repose sur deux concepts de base :

- L'abstraction, notée $\lambda x.e$, qui correspond à la définition de fonction
- et l'application, notée $(e_1\ e_2)$, qui correspond à l'appel de la fonction e_1 en lui passant en paramètre e_2.
- à cela il faut ajouter une infinité de symboles : x, y, z, ...
- et une règle de réécriture : la β-reduction :

$$(\lambda x.e_1\ e_2) \rightarrow_\beta e_1[x := e_2]$$

Exemple : bulletin météo

- On va utiliser comme symboles des mots comme : aujourd'hui, demain, il, fera, beau, chaud, etc.
- Exemple d'expression en forme normale :

$$(((\text{aujourd'hui il}) \text{ fera}) \text{ beau})$$

- La même sans toutes les parenthèses :

$$(\text{aujourd'hui il fera beau})$$

- Exemple d'abstraction en rendant variable *aujourd'hui* :

$$\lambda \textit{aujourd'hui}.(\textit{aujourd'hui} \text{ il fera beau})$$

Lambda Calcul

Exemple : bulletin météo

- Exemple d'application :

$$(\lambda \textit{aujourd'hui}.(\textit{aujourd'hui}\ \text{il fera beau})\ \text{demain})$$

- Exemple de β-reduction :

$$\frac{(\lambda \textit{aujourd'hui}.(\textit{aujourd'hui}\ \text{il fera beau})\ \text{demain})}{(\text{demain il fera beau})}\quad(\beta)$$

Lambda Calcul

Exemple : bulletin météo

- générateur de bulletins météos à deux variables

$$\lambda \textit{aujourd'hui}.\lambda \textit{beau}.(\textit{aujourd'hui}\ \text{il fera}\ \textit{beau})$$

- et son application à demain et chaud

$$\frac{\underline{(\lambda \textit{aujourd'hui}.\lambda \textit{beau}.(\textit{aujourd'hui}\ \text{il fera}\ \textit{beau})\ \text{demain}}\ \text{chaud})\quad(\beta)}{\underline{(\lambda \textit{beau}.(\text{demain il fera}\ \textit{beau})\ \text{chaud})}\quad(\beta)}$$
$$(\text{demain il fera chaud})$$

Lambda Calcul

En fait le λ-calcul ne comporte aucune des notions habituelles de l'informatique :

- ni les nombres
- ni les booléens
- ni les structures de données
- ni les structures de contrôle
- ni même la récursivité

Toutes ces notions peuvent être construites à partir des seules notions d'abstraction et d'application !

Représentation des nombres

$$
\begin{aligned}
0 &= \lambda f.\lambda x.x \\
1 &= \lambda f.\lambda x.(f\ x) \\
2 &= \lambda f.\lambda x.(f\ (f\ x)) \\
&\dots \\
\text{succ} &= \lambda n.\lambda f.\lambda x.(f\ (n\ f\ x)) \\
\text{plus} &= \lambda m.\lambda n.\lambda f.\lambda x.(m\ f\ (n\ f\ x)) \\
\text{mult} &= \lambda m.\lambda n.\lambda f.\lambda x.(m\ (n\ f)\ x) \\
\text{exp} &= \lambda m.\lambda n.\lambda f.\lambda x.(n\ m\ f\ x)
\end{aligned}
$$

Représentation de la récursivité

$$\mathsf{Y} \quad = \quad \lambda b.(\lambda v.(b\ (v\ v))\ \lambda v.(b\ (v\ v)))$$

Lambda Calcul

Malgré sa simplicité le λ-calcul est d'une importance capitale car il fonde la notion de calculabilité.

- Thèse de Church : toute fonction effectivement calculable peut être exprimée en λ-calcul
- Equivalence entre λ-calcul, machines de Turing, fonctions récursives.
- Influence considérable sur les langages de programmation et la théorie des LP
- Impact philosophique, vision "fonctionnelle" des choses.

Lambda Calcul

Le lambda-calcul est un langage de programmation réduit à son essence. Cette essence peut servir de base à de nouveaux langages de programmation comme le montrent les deux exemples suivants :

- Graphic-calculus
- Elody

Logique Combinatoire

Logique combinatoire

- La logique combinatoire a été inventée par Moses Shönfinkel en 1920 et publié en 1924 dans un article intitulé “On the building blocks of mathematical logic”.
- L’idée de Shönfinkel était de voir si l’on pouvait se passer de la notion de variable, qu’il considérait comme un concept auxiliaire.

Définition avec variable

Voici un exemple de définition de fonction utilisant des variables :

$$\text{dist}(x_0, y_0, x_1, y_1) = \sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2}$$

Substitution

Au moment de l'utilisation de la fonction $\text{dist}(2, 20, 7, 1)$ on remplace dans sa définition les variables x_0, x_1, y_0, y_1 par leurs valeurs respectives $2, 20, 7, 1$.

$$\begin{aligned} \text{dist}(2, 20, 7, 1) &= \sqrt{(2 - 7)^2 + (20 - 1)^2} \\ &= \sqrt{-5^2 + 19^2} \\ &= 19.65 \end{aligned}$$

Logique combinatoire

Moses Shönfinkel a montrer que l'on pouvait se passer de fonctions à plusieurs variables et se contenter de fonctions à une variable !

Curryfication

Une fonction à plusieurs variables pouvait être représentée par un emboitement de fonctions à une variable (ce que l'on appelle aujourd'hui la curryfication)

$$\mathrm{F}(x,y) \rightarrow (\mathrm{F}(x))(y)$$

et de fait on peut se passer des parenthèses :

$$(\mathrm{F}(x))(y) \rightarrow (\mathrm{F}\ x)\ y$$
$$(\mathrm{F}\ x)\ y \rightarrow \mathrm{F}\ x\ y$$

Logique combinatoire

Exemple de curryfication

$$\begin{aligned} \text{transposer}(12,S) &\rightarrow \text{transposer } 12\ S \\ \text{transposer } 12 &\rightarrow \text{octavier} \end{aligned}$$

Parenthèses implicites

$$\text{transposer } 12\ S \leftrightarrow ((\text{transposer } 12)\ S)$$

Logique combinatoire

Mais surtout Moses Shönfinkel a montré que l'on pouvait totalement se passer de variables. Il a proposé pour cela un jeux initial de 5 combinateurs :

B, C, S, K et I

$$
\begin{aligned}
\mathsf{B}\ x\ y\ z &\rightarrow x\ (y\ z)\\
\mathsf{C}\ x\ y\ z &\rightarrow x\ z\ y\\
\mathsf{S}\ x\ y\ z &\rightarrow (x\ z)\ (y\ z)\\
\mathsf{K}\ x\ y &\rightarrow x\\
\mathsf{I}\ x &\rightarrow x
\end{aligned}
$$

Logique combinatoire

Il a ensuite montré que l'on pouvait se contenter de 3 combinateurs

S K et I

$$\begin{array}{rcll} \mathsf{S}\ x\ y\ z & \to & (x\ z)\ (y\ z) & (\mathrm{c1}) \\ \mathsf{K}\ x\ y & \to & x & (\mathrm{c2}) \\ \mathsf{I}\ x & \to & x & (\mathrm{c3}) \end{array}$$

et même que I n'était pas nécessaire !

$$\mathsf{I} \quad = \quad (\mathsf{S}\ \mathsf{K}\ \mathsf{K})$$

Démonstration

$$\begin{array}{rr} \mathsf{S}\ \mathsf{K}\ \mathsf{K}\ x & (\mathrm{c1}) \\ \mathsf{K}\ x\ (\mathsf{K}\ x) & (\mathrm{c2}) \\ x & (\text{arrêt}) \end{array}$$

Logique combinatoire

En résumé Moses Shönfinkel a montré que l'on pouvait exprimer tout calcul (et donc toute l'informatique qui n'existait pas encore) à l'aide de deux combinateurs

S et K

$$\begin{array}{rcll} \mathsf{S}\ x\ y\ z & \rightarrow & (x\ z)\ (y\ z) & (\mathrm{c1}) \\ \mathsf{K}\ x\ y & \rightarrow & x & (\mathrm{c2}) \end{array}$$

On ne peut pas faire plus concis !

Logique combinatoire

Pour montrer que la logique combinatoire est aussi expressive que le lambda-calcul il suffit de montrer comment convertir toute expression du lambda-calcul en logique combinatoire.

Trois règles de réécriture suffisent !

$$
\begin{array}{rcll}
\lambda x.x & \rightarrow & \mathsf{I} & (\mathrm{r1}) \\
\lambda x.c & \rightarrow & (\mathsf{K}\ c) & (\mathrm{r2}) \\
\lambda x.(e_1 e_2) & \rightarrow & (\mathsf{S}\ (\lambda x.e_1)\ (\lambda x.e_2)) & (\mathrm{r3})
\end{array}
$$

Logique combinatoire

Voici un exemple de conversion d'une expression du λ-calcul en logique combinatoire en appliquant les règles de réécriture précédentes.

Conversion de $\lambda x.\lambda y.y$

$$\begin{array}{ll} \lambda x.\underline{\lambda y.y} & \text{(r1)} \\ \underline{\lambda x.\mathsf{I}} & \text{(r2)} \\ \mathsf{K}\ \mathsf{I} & \text{(arrêt)} \end{array}$$

Vérification

$$\begin{array}{ll} \underline{\mathsf{K}\ \mathsf{I}\ x}\ y & \text{(c2)} \\ \underline{\mathsf{I}\ y} & \text{(c3)} \\ y & \end{array}$$

Logique combinatoire

Utilisations de la logique combinatoire :

- Implementation de langages fonctionnels
- Machine S K de Turner
- langages FP (Backus, 1977)
- Faust (Orlarey, Fober, Letz 2002)

Conclusion

Au fond, que reste-t-il ?

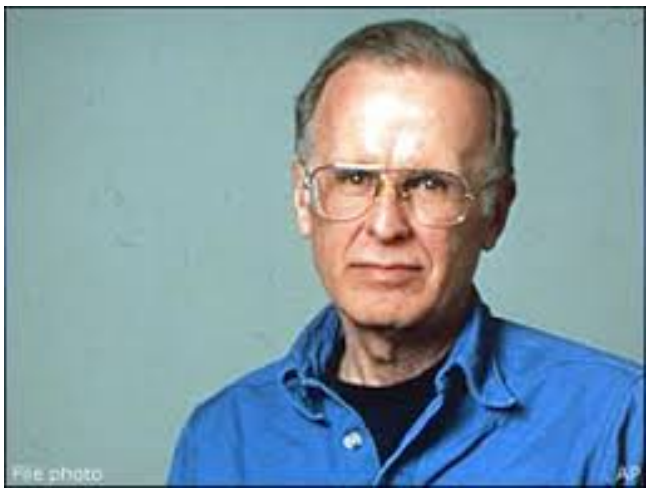
Quand on a enlevé tout le superflu il ne reste plus grand chose !

- la plupart des notions : nombres, booléens, structures de données, structures de contrôle ont disparu ;
- et même la notion de variable et de récursivité ;
- il ne reste que la notion de **fonction**.

L'essence d'un langage de programmation c'est de permettre de décrire des fonctions. Tout le reste peut se construire par dessus.

John Backus, ACM Turing Award 1977

Can Programming Be Liberated from the von Neumann Style ? A Functional Style and Its Algebra of Programs



"Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak :. . . "