



Benoît Sonntag

Benoît Sonntag, maître de conférences à l'Université de Strasbourg, est l'auteur du langage Lisaac. Il est expert en compilation et en programmation système. Grâce à son expertise, le compilateur Lisaac Ω atteint des performances comparables, voire supérieures, à C ou C++.



Dominique Colnet

Dominique Colnet, professeur à l'IUT Nancy Charlemagne de l'Université de Lorraine, est membre du LORIA. Il est l'auteur de SmartEiffel, le compilateur GNU pour le langage Eiffel. Il est également spécialiste des langages de programmation, de la compilation et du génie logiciel. Il collabore depuis plusieurs années avec Benoît Sonntag pour le développement de Lisaac Ω .

Lisaac Ω : un langage visuel, typé et performant

Les langages informatiques ont connu une évolution spectaculaire entre 1960 et 1990, période durant laquelle la plupart des concepts clés de la programmation moderne ont émergé: gestion de la mémoire, compilation, fusion code-données, programmation fonctionnelle ou programmation par objets.

Cependant, depuis lors, l'innovation syntaxique et la créativité dans les langages ont marqué le pas. Les notations orientées C dominent encore et les grammaires hors-contexte ont atteint leurs limites, creusant l'écart entre le matériel moderne et les environnements de programmation.

À l'ère du graphisme et du multimédia en général, il est temps qu'un programme ne ressemble plus à une suite d'instructions utilisant des syntaxes obscures... **Figure 1**

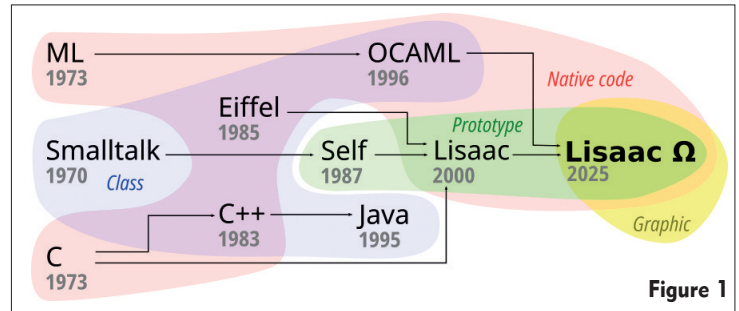


Figure 1

C'est dans ce contexte que Lisaac Ω a été conçu.

Même si nous sommes encore loin de Tony Stark en train de concevoir avec son cher JARVIS, Lisaac Ω est un langage complètement novateur qui franchit une étape.

Il propose une refonte complète de la représentation et de l'édition du code, conciliant expressivité, performance et approche visuelle, tout en restant fidèle à des principes rigoureux hérités d'Eiffel, comme le *typage* et la *programmation par contrat*. **Figure 2**

Contrairement aux langages traditionnels, Lisaac Ω ne se limite pas à un simple fichier texte soumis à une grammaire hors contexte. Chaque programme est représenté comme une figure éditable, manipulable grâce à l'outil intégré *Elit*. Cette approche permet une visualisation intuitive et naturelle. Le code devient un document visuel avec des images, des animations, des équations, des hyperliens...

Le langage supporte pleinement les notations mathématiques, permettant d'utiliser des symboles classiques pour les types et les opérations, tout en conservant une distinction claire entre comparaison et affectation (cf. le fameux problè-

me de '=' et '==' du C repris dans de nombreux langages dits « modernes »). La représentation graphique personnalisable offre une flexibilité inédite pour les domaines spécifiques, comme la robotique, la physique ou l'électronique.

Le mathématicien ou le physicien n'a plus à traduire son langage métier vers un langage informatique complexe. C'est le langage informatique qui s'adapte à la terminologie de son métier.

Lisaac Ω est un langage fortement typé, prenant en charge la programmation par contrat, les fonctions anonymes, la manipulation de blocs de code et les opérateurs définis par l'utilisateur.

Il s'adresse aussi bien aux grands projets industriels qu'au prototypage rapide, et parfaitement à l'enseignement !

Visualiser l'héritage et comprendre la structure des objets - Figure 3

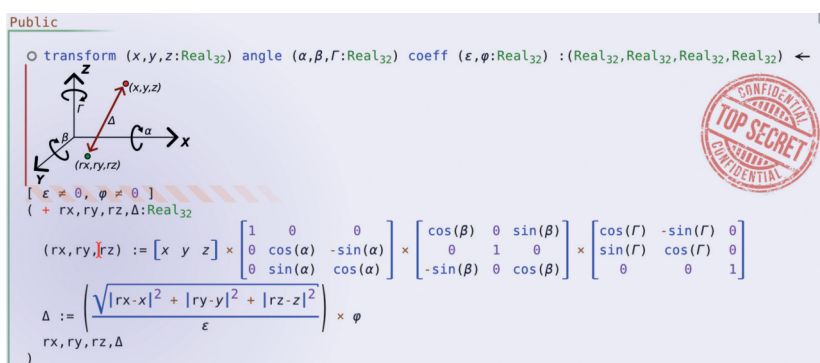
C'est par une schématisation graphique que l'on précise interactivement avec *Elit* les relations d'héritage entre les objets.

C'est également grâce à un schéma généré automatiquement, en fonction des attributs hérités, que l'on peut vérifier la structure des objets. La figure *inherit-string-buffer* montre la visualisation réalisée par *Elit* du graphe d'héritage de *String_buffer*, ainsi que de la structure de l'objet correspondant.

Hello World : direct, simple, clair, visuel - Figure 4

Rien de plus direct que le traditionnel programme *Hello World*, présenté ici en Lisaac Ω . Pas besoin d'*include* ni d'*import*, tout est immédiatement prêt à l'emploi. L'exemple montre la simplicité et la clarté du langage: chaque instruction est visible, intuitive, et entièrement manipulable dans l'éditeur visuel *Elit*, lui-même complètement conçu en Lisaac Ω .

Figure 2



Parenthèses facultatives : une syntaxe réellement flexible **Figure 5**

Le code précédent reste parfaitement valide sans parenthèses. De même, la séparation des instructions peut se faire avec ou sans point-virgule. *Lisaac* Ω privilégie une syntaxe souple, adaptée au contexte et au style du développeur, sans rigidité inutile.

Choisissez votre style d'écriture **Figure 6**

L'instruction conditionnelle existe sous plusieurs formes, qui peuvent cohabiter naturellement dans un même programme. Les amateurs de *Smalltalk* y retrouveront une écriture familière, les adeptes d'une notation plus classique conserveront leurs repères, et les débutants pourront utiliser la version explicite en français du *si-alors-sinon*. Chacun peut écrire selon sa culture, sans contrainte imposée par le langage. Cette flexibilité est rendue possible par le simple fait que la conditionnelle n'est pas une brique de base du langage; elle est définie en bibliothèque.

Maîtriser l'héritage et le polymorphisme visuellement ! **Figure 7**

L'éditeur *Elit* présente de manière graphique les relations d'héritage pour chaque objet. En éditant **Boolean** (cf. figure *inherit-boolean*), il est possible de visualiser à la fois tous ses ancêtres mais aussi tous les descendants de **Boolean**. Chaque lien d'héritage peut être configuré soit en trait plein, soit en trait pointillé.

En trait plein, l'héritage est complet, incluant tous les slots, avec autorisation d'affectation.

En trait pointillé, l'héritage est tout aussi complet, mais rejette la possibilité d'affectation. C'est ce que l'on appelle l'héritage *alimentaire* ou encore l'héritage non-conforme.

Une affectation n'est possible entre deux types que s'il existe un chemin en trait plein reliant ces deux types. En conséquence, il est correct d'affecter **True** dans **Boolean**, mais aucune affectation de **Boolean** n'est possible dans **Print** ou dans **Utils**. Cette représentation garantit que le type **Boolean** conserve exactement ses deux uniques valeurs. En outre, comme **Clone** n'est pas dans l'arbre, **True** et **False** sont des singletons.

Un if entièrement défini par les objet ou quand le booléen décide ! **Figure 8**

Grâce à l'utilisation de blocs de code en paramètre, *Lisaac* Ω permet de définir toutes les structures de contrôle directement comme des méthodes dans les objets, aussi appelés prototypes. La figure *if-then-else* illustre la définition de l'instruction **if-then-else** dans le prototype **Boolean**.

Comme le montre l'éditeur *Elit*, la méthode abstraite est ensuite redéfinie dans les prototypes **True** et **False**, assurant un comportement spécifique selon la valeur dynamique du booléen **self** testé. Les rectangles **True** et **False** sont des hyperliens vers les redéfinitions respectives.

Héritage multiple et visualisation de la structure des objets **Figure 9**

La figure *inherit-array* illustre la visualisation de l'héritage réalisée par *Elit* pour le type **Array(V)**. Un objet peut hériter

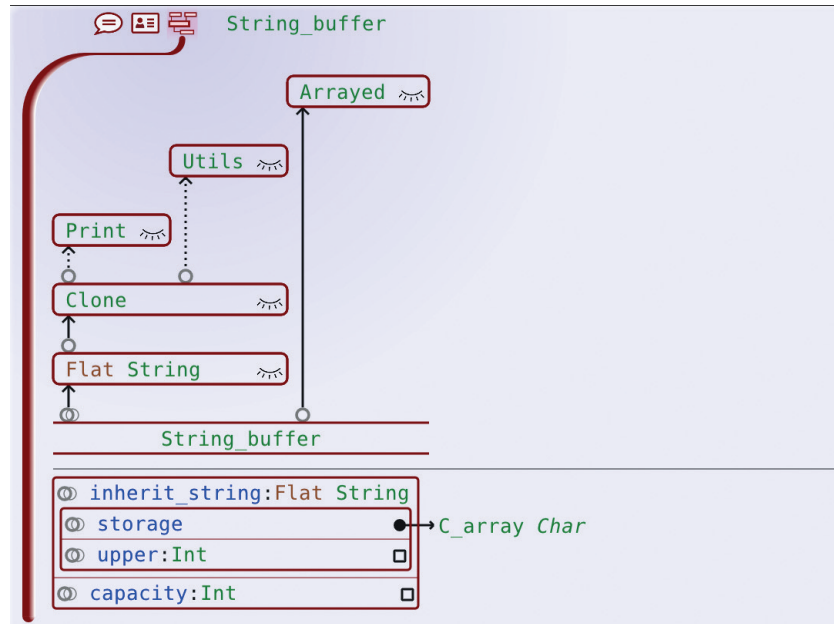


Figure 3

```
o main ← print("Hello world!\n")
```

Figure 4

```
o main ←
| This is a comment into the main procedure header.
(
  print("Parentheses" + "are")
  print "optional, and so"
  print("is" + "the") ; print "semicolon"
  print("for separating") ;
  print "two statements" // Comment inside code
)
```

Figure 5

```
(a = 0) if { // SmallTalk/Object style
  "Zero!" println
}
if (a = 0) then { // Standard
  print "Zero!\n"
}
si (a = 0) alors { // French
  écrire "Zero!\n"
}
```

Figure 6

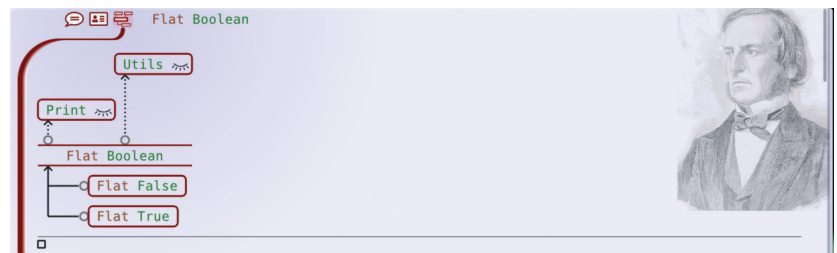


Figure 7

Conditional :

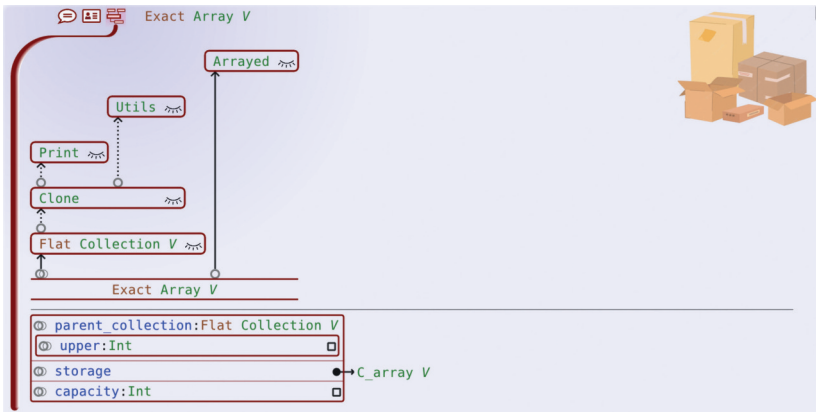
```
o if self:Self then tblk:{} else fblk:{} :Boolean ← abstract
[Flat False] [Flat True]
```

Figure 8

d'un ou plusieurs objets, sans restriction sur la nature de l'héritage. La mention **Exact** sur **Array** indique qu'une variable de ce type ne peut contenir que des objets de cette catégorie.

En outre, *Elit* affiche la structure de l'objet en mémoire. La figure montre clairement l'ordre de recherche des slots, de bas en haut, depuis **Array(V)**. Rien n'est laissé au hasard : tout est présenté de manière visuelle et compréhensible.

Figure 9



```

o self:Self '~=' Left 80  => other:Self ←
(
  |self - other| ≤ 0.1
)

```

Figure 10

```

o age:Int ←
| First call is computed; subsequent calls are simple reads.
(
  age := today - birth_day
)

o hi ←
| Code replacement on the first call.
(
  "Hello Stranger!" println
  hi ← ( "Hello Buddy!" println )
)

```

Figure 11

```

o array_loop_démo ←
| Using a local variable arr of type Array Int:
( + arr:Array Int
  arr := new Array Int of (1, 2 , 3, 4)

  arr.add_last 5 // adding 5 at the end:

  arr.add_last (6, 7) // now adding 6 then 7

  arr.foreach {v:Int ; println(v) }

  // Another way to print all values inside array arr:
  from 0 to (arr.upper) do { idx:Int
    (arr.at idx) println
  }
)

```

Figure 12

```

o from self:Self to limit_up:Self do blc:{Self;} ←
(
  if (self ≤ limit_up) then {
    blc.value self
    from (self + 1) to limit_up do blc
  }
)

```

Figure 13

Lisaac Omega offre de multiples formes d'héritage : un équivalent de l'héritage de classe, ou encore l'héritage à trait (comme en Rust).

Nous avons aussi l'héritage dynamique ou l'héritage calculé.

Opérateurs sur-mesure : flexibilité et puissance - Figure 10

Dans *Lisaac Omega*, on peut créer autant d'opérateurs que l'on souhaite (à l'exception des symboles d'affectation qui sont réservés). Cela inclut les opérateurs arithmétiques, les comparateurs et même l'opérateur élémentaire **=** qui peut être adapté pour chaque catégorie d'objet. Pour les opérateurs binaires, comme l'opérateur **~=** illustré dans la figure *operator-def*, la définition précise leur priorité (ici, 80), leur associativité (gauche ou droite) et la commutativité des types.

Cette flexibilité permet d'ajuster les comportements des opérateurs au plus près des besoins du domaine et du code utilisateur, tout en conservant une syntaxe claire et rigoureuse. Il est également possible de spécifier une évaluation paresseuse des arguments, ce qui est particulièrement utile pour la définition des opérateurs logiques comme **&&** et **||**.

Une telle flexibilité au niveau de la définition des opérateurs est tout à fait inédite !

Pour compléter les opérateurs binaires, nous avons aussi les opérateurs unaires préfixés (comme **-** par exemple) ou les opérateurs unaires post-fixés (comme **!**, pour la factorielle par exemple).

Attributs ou méthodes : la fin de la distinction... - Figure 11

Un slot peut recevoir soit une valeur simple, remplaçant directement le code existant (voir le slot **age** dans la figure *power-hi-age*), soit un nouveau bloc de code complet (exemple pour le slot **hi** dans la même figure).

En *Lisaac Omega*, tout est dynamique !

Une généricité sans exception : tous les types sur un pied d'égalité - Figure 12

L'exemple de la figure *array-loop-demo* montre l'utilisation du type **Array(Int)** dans lequel **V** représente un paramètre de type quelconque. Contrairement à Java, même les types de base sont des objets.

Lisaac Omega offre, comme le langage *Eiffel*, un mécanisme de programmation par contrats avec des pré et postconditions, défini au niveau abstrait et hérité dans tous les sous-types. Le système de typage est encore plus performant que celui d'*Eiffel* et ne comporte aucune faille.

Notons en outre que *Lisaac Omega* permet la définition de méthode ayant un nombre variable de paramètres: dans l'exemple, la même fonction **add_last** peut être appelée avec un seul ou plusieurs arguments. C'est à la fois plus concis, mais aussi plus efficace à l'exécution.

Choisir ses propres structures de contrôle.- Figure 13

Lisaac Omega permet de définir de nouvelles structures de contrôle-

le directement dans la bibliothèque d'objets, sans les figer dans le compilateur. La figure *from-to-do* illustre un exemple : une boucle personnalisée définie comme méthode sur les entiers, l'entier correspondant étant ici placé après le mot-clé **from**.

Dans cet exemple, l'appel récursif est un appel terminal. Le compilateur remplace alors automatiquement cet appel par une boucle itérative. Cette transformation garantit une exécution optimale : la boucle générée est équivalente à ce que l'on pourrait écrire en C.

Grâce à cette flexibilité, *Lisaac Ω* offre également aux débutants la possibilité de programmer directement dans leur langage.

Par exemple, la bibliothèque fournie en standard permet d'utiliser une notation française pour les cours d'algorithmique, simplifiant ainsi l'apprentissage tout en conservant la rigueur et les performances du langage.

Ainsi, *Lisaac Ω* combine la flexibilité d'un langage haut niveau avec l'efficacité maximale d'une exécution bas niveau, tout en offrant à l'utilisateur un contrôle total sur ses structures de programmation.

ℤ natif: les types mathématiques dans le langage - Figure 14

Le code de la fonction **factorial** correspond à l'écriture récursive classique, telle qu'elle est souvent présentée aux débutants. L'exemple montre qu'il est possible d'utiliser directement le type $\mathbb{Z}$, sans bibliothèque externe ni artifice syntaxique.

Dans cet exemple, l'argument est écrit sans parenthèses, mais cette notation reste optionnelle. Une écriture parenthésée est tout aussi valide, aussi bien pour la définition que pour l'appel. Le langage ne contraint pas inutilement le développeur à une unique forme syntaxique. Tant qu'aucune ambiguïté n'est introduite, le compilateur de *Lisaac Ω* reste tolérant et laisse à l'utilisateur le choix de la notation qui lui semble la plus naturelle.

Cette souplesse apparente repose en réalité sur une avancée technologique majeure : la compilation devient contextuelle et ne repose plus exclusivement sur des grammaires LR ou LALR comme c'est traditionnellement le cas.

À ce titre, *Lisaac Ω* marque une rupture significative avec l'approche syntaxique des langages classiques, en conciliant liberté d'écriture et rigueur formelle.

Fonctions à résultats multiples: une simplicité naturelle. - Figure 15

La figure *fibonacci-z* illustre la possibilité native dans *Lisaac Ω* de définir des fonctions retournant plusieurs résultats.

Dans cet exemple, la fonction **fibonacci** repose sur une fonction auxiliaire récursive, **fiborec**, qui renvoie simultanément deux valeurs de type $\mathbb{Z}$.

Cette approche permet une définition récursive à complexité linéaire, garantissant ainsi une efficacité optimale. Plutôt que de recalculer inutilement des valeurs intermédiaires, les deux résultats nécessaires sont propagés ensemble, de manière explicite et typée.

L'utilisation du type $\mathbb{Z}$ garantit par ailleurs une précision arbitraire, sans limitation liée à la taille des entiers machine. Les

```

○ factorial n:ℤ : ℤ ←
| Simply use the ℤ type if you prefer unlimited values.
( + result:ℤ
  if (n ≤ 1) then {
    result := 1
  } else {
    result := n × factorial (n - 1)
  }
  result
)

```

Figure 14

```

○ fiborec n:ℤ : (ℤ, ℤ) ←
| Still recursive, but with two results.
( + f0, f1:ℤ
  if (n ≤ 1) then {
    (f0,f1) := (0,1)
  } else {
    (f0,f1) := fiborec (n-1)
    (f0,f1) := (f1, f0+f1)
  }
  (f0,f1)
)

○ fibonacci n:ℤ : ℤ ←
( + f1, f2:ℤ
  (f1,f2) := fiborec n
  f2
)

```

Figure 15

```

// Matrix:
a :=  $\begin{bmatrix} 2 & 1 \\ 4 & 7 \end{bmatrix} \times \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 6 & 7 \end{bmatrix}$ 

// Discriminant Δ of the quadratic equation:
Δ :=  $\frac{ax + by + c}{\sqrt{a^2 + b^2}}$ 

// Lemniscate of Bernoulli:
x :=  $\frac{\cos(tb)}{1 + \sin(tb)^2}$  ; y :=  $\frac{\cos(tb) \times \sin(tb)}{1 + \sin(tb)^2}$ 

// Flower:
x :=  $\sqrt{|\cos(3 \times t)|} \times \cos(t)$  ; y :=  $\sqrt{|\cos(3 \times t)|} \times \sin(t)$ 

// Lisaajous:
x :=  $\sin\left(3 \times t + \frac{\pi}{4}\right)$  ; y :=  $\cos(2 \times t)$ 

harmonic :=  $\sum_{k=1}^n \left\{ \frac{1}{k} \right\}$ 

```

Figure 16

calculs peuvent ainsi croître sans perte d'exactitude. Il est important de souligner que l'usage d'une fonction à résultats multiples, comme **fiborec**, n'entraîne aucune allocation mémoire supplémentaire. Les valeurs sont retournées directement, sans création d'objets intermédiaires inutiles.

Lisaac Ω a été conçu pour offrir les meilleures performances, aussi bien en termes de vitesse d'exécution que de consommation mémoire, tout en conservant une écriture claire et rigoureuse.

Les mathématiques comme on a envie de les voir ! - Figure 16

La figure *maths* présente un ensemble de notations mathématiques usuelles déjà définies et directement utilisables

```

○ half_adder (a, b: Boolean) : (Boolean, Boolean) ←
( + sum: Boolean
+ carry: Boolean
sum := 
carry := 
(sum, carry)
)

```

Figure 17

```

○ full_adder (a, b: Boolean) carry carry: Boolean : (Boolean, Boolean) ←
( + xor_ab: Boolean
+ sum: Boolean
xor_ab := 
(  ,  )
)

```

Figure 18

dans le code des utilisateurs. On y retrouve notamment l'écriture des matrices, la racine carrée, la mise à la puissance, le quotient, les opérateurs trigonométriques, la valeur absolue, ainsi que l'opérateur de somme avec sa présentation traditionnelle. Toutes ces notations sont affichées en vectoriel, ce qui permet d'obtenir un rendu clair, précis et fidèle aux conventions mathématiques enseignées.

Le code se rapproche ainsi des écritures utilisées par les enseignants, réduisant l'écart entre le formalisme mathématique et son implémentation informatique. L'édition de ces formes graphiques est particulièrement simple grâce à l'outil *Elit*.

Il n'est pas nécessaire d'utiliser des balises complexes ni de naviguer dans des menus lourds. L'édition repose sur un principe contextuel : lorsque le curseur s'approche d'une zone modifiable, l'affichage bascule automatiquement vers une notation textuelle simple, permettant une modification directe et naturelle.

Toutes les opérations peuvent être réalisées au clavier, sans recours obligatoire de la souris. Il est important de souligner que l'éditeur *Elit* est lui-même entièrement programmé en *Lisaac Ω*, garantissant ainsi une portabilité complète et une cohérence totale entre le langage et son environnement de développement.

Du circuit à l'écran

Il est également possible d'ajouter de nouvelles formes graphiques au langage.

Ces formes permettent, par exemple, comme l'illustrent les figures *half-adder* et *full-adder*, de représenter directement des portes logiques à l'intérieur même du code.

Figures 17 et 18

Dans la figure *half-adder*, le demi-additionneur calcule la somme et la retenue en utilisant la version graphique des opérateurs AND et XOR. La structure du schéma correspond exactement à la structure logique du calcul : la somme est obtenue par un XOR entre les deux entrées, tandis que la retenue résulte d'un AND. Le code devient ainsi une représentation visuelle fidèle du circuit logique. Dans la figure *full-adder*, l'additionneur complet étend ce principe en combi-

nant plusieurs opérations logiques. Les deux valeurs booléennes constituant le résultat de la méthode sont intégrées directement dans le code, à la position même des résultats. Cette intégration rend explicite la correspondance entre la construction graphique et la sémantique de la méthode. Tous ces éléments graphiques sont disponibles au sein d'une bibliothèque graphique extensible.

Comme nous l'avons vu, cette bibliothèque prend déjà en charge la majorité des notations mathématiques usuelles. Dès qu'une nouvelle visualisation est ajoutée, elle devient immédiatement utilisable dans les objets définis par l'utilisateur.

Une correspondance systématique entre forme graphique et notation objet permet ainsi d'intégrer naturellement ces graphismes dans le langage, sans perdre la compatibilité avec les outils classiques de gestion du code, tels que Git.

Des performances vraiment à la hauteur

Le compilateur *Lisaac Ω* analyse le programme dans sa globalité pour prédire les types et spécialiser le code, résolvant statiquement la grande majorité des appels de méthodes dynamiques. Le résultat : des binaires natifs compacts et extrêmement efficaces — la preuve que la programmation basée sur les prototypes et les hautes performances ne sont pas incompatibles.

Lisaac Ω est le premier langage « tout objet » offrant de telles performances, rivalisant avec du C ou C++ écrits à la main.

Une portabilité native – Fonctionne vraiment partout

Là où de nombreux langages comme *Java* ou *Python* se reposent sur des bibliothèques C efficaces pour des tâches critiques, comme le graphisme, *Lisaac Ω* est indépendant. Cela n'a été rendu possible que par ses excellentes performances et sa capacité à faire de la programmation de bas niveau. Indépendant de son environnement, cette approche atypique lui confère une portabilité native exceptionnelle. Le même code fonctionne sans modification sous *Linux*, *macOS*, *Windows* et même *Android*.

Il peut également s'exécuter directement dans un navigateur Web via le *WebAssembly*, et même gérer des microcontrôleurs de type *Arduino* !

Testé et approuvé par l'Université de Lorraine

Depuis septembre 2025, *Lisaac Ω* est utilisé en conditions réelles à l'IUT Nancy Charlemagne de l'Université de Lorraine (<https://iut-charlemagne.univ-lorraine.fr/>).

Ce site pilote va monter en puissance avec l'ensemble des étudiants de première année dès la rentrée 2026 au sein du département informatique. Le cours d'initiation à l'algorithmique sera entièrement réalisé à l'aide de vidéos.

Rendez-vous sur lisaac.org pour télécharger *Lisaac Ω*, accéder aux articles de recherche et découvrir ses vidéos de présentation.

Site : lisaac.org