

Future Language TV ( 2009-08-30 )

# LUCIFERの設計コンセプトと 導入予定の機能紹介

# 自己紹介(1/2)

- 名前：Wraith the Trickster / 道化師
- 活動：主にC++界隈でくだをまいている。
- 特徴：一人ガラパゴス諸島。
- アドレス：
  - Mail：Spam受信ツールと化してるので・・・
  - Twitter：<http://twitter/wraith13/>
  - Blog：<http://d.hatena.ne.jp/wraith13/>
  - Site(info)：<http://www.trickpalace.net/>
  - Site(lib)：<http://tricklib.com/>

# 自己紹介(2/2)

- 主義：正しさよりも個性重視
  - 正しさが絶対だとするならばそもそも個である意味もなければ，人である意味もない。
  - それが例え間違っていることだとしても自分が感じたことや自分が考えたことを重視したい。
- 誰よりも怪気炎を放てたら勝ちだと思ってる。
- 結局の所，この自己紹介で何が言いたいかと言うと「自分の言うことを鵜呑みにしないでください」。

# 概要

- 未来の言語の前に
  - 未来について
  - 知能について
  - 生物について
  - 未来のコンピューティング
- 設計コンセプト
  - 一般論としてどのような言語を設計すべきか
  - Luciferとしてはどのように設計するのか
- 導入予定の機能紹介
  - 豊富な型修飾 ( plenty type qualifications )
  - 型復元/修飾復元 ( type restore / qualification restore )
  - 完全演算表示 ( providence )
  - 超例外処理 ( super exception handling )
  - 連鎖参照 ( presenter )

Luciferの設計コンセプトと導入予定の機能紹介

未来の言語の前に  
未来について

# 未来について(1/3)

- 残されてる時間はそんなにない
  - 早ければ半世紀も経たない内に人の歴史は何らかの形で終わる.
  - 変化の速さは加速する一方で行き着くところまで言ってしまふのにもうそんなに時間は残っていない.
  - 行き着くところまで行ってもたいしたことのない、とってもショボイ未来である可能性もある.
  - もちろん、その前に戦争や病気などで人類が滅んでしまう可能性もある.
  - 中途半端な戦争や不景気によって停滞するのが心配.

# 未来について(2/3)

- 任意の立場から見た任意の変化について
  - そのほとんどが観測できない，もしくは影響がない。
  - 影響のある変化というのはそのほとんどが災いであり，幸運な変化というものは稀。
    - 例えば，突然事故などで愛車を失う確率と，宝くじにでも当たって愛車を手にする確率には絶望的なまでの差がある。
  - 変化が激しくなるこれから時代においてその変化をコントロールできない立場にいることはおそらく地獄をみることを意味する。

# 未来について(3/3)

- 予想されうる未来の姿・出来事
  - ポストヒューマン(2045年).
  - コンピュータを使った思想攻撃・洗脳.
    - 軽度の行動誘導なら実はもう既に株価操作SPAMメールという実績がある.



人工知能界隈の技術開発は  
急務であり且つ生存をかけた戦い

Luciferの設計コンセプトと導入予定の機能紹介

# 未来の言語の前に 知能について

# 知能について(1/11)

- 体力と知力の特性の違いについて
  - 体力的に優れた人間の代わりは多少の問題に目を瞑れば凡夫でもどうにかなるが、知力的に優れた人間の代わりはそうはいかない。

# 知能について(2/11)

- 体力的に優れた人間の代替の例.
  - 重い者を持ち上げることができる人間の代わりは人数を割くだけでどうにでもなる.
  - 速度を競う競技であれば、余計に時間がかかるだけで誰でもいつかはゴールに辿り着く.
  - ルールに少々目を瞑れば素人を100人でも投入すれば優れたサッカー選手の代わりが十分にとまる.
  - 反例：泳げない者を100人集めても泳げる者の代替は勤まらない。→ただし訓練により可能になる.
- 限定条件がなければ体力的に優れた人間の代替は概ねどうとでもなる.

# 知能について(3/11)

- 知力的に優れた人間の代替の例.
  - 英語が話せない人間を100人集めても英語での会話はできない。→ただし訓練により可能になる。
  - いくら時間をかけてもポイントを理解できなかった者を何人集めようが、さらに時間的にもいくらでも目を瞑ろうが、ポイントを理解しているように振る舞うことはできない。
- 知力的に優れた人間の代替は難しい。

# 知能について(4/11)

- なぜ，体力と知力でこのような差がうまれるのか？
- 恐らく，次の二つの要素が鍵
  - ワーキングメモリ
  - 思考の指向性

# 知能について(5/11)

## ■ ワーキングメモリ

- ワーキングメモリは短記憶と呼称されることもあり，CPUで言うレジスタのようなもので，個人差があり一般的に5～9つの情報を記憶できる。
- ワーキングメモリが優れていると複雑なコンテキストの情報も容易く理解できる。
- 引数の数は最大でも7つまでとされるのもこのワーキングメモリの容量の平均値に由来。
- 腕力と違って一人で理解できない(ワーキングメモリが足りない)ものを複数人で協力して理解に至る・・・なんてことはできない。

# 知能について(6/11)

- 思考の指向性
  - このスイカの写真をみてください。



- <http://ja.wikipedia.org/wiki/スイカ>より引用

# 知能について(7/11)

- 思考の指向性
  - いまのスイカを見てなにを思い浮かべました？
    - スイカの精確な品種？
    - スイカの熟れぐあい？
    - スイカの味？
    - スイカにまつわるなにかの思い出？
  - 全く同じ入力情報にも関わらず，その出力は人によってあまりにもてんでバラバラ。
  - ある天才的な人物の思考の代わりとするには不利な特性かもしれないが，複数人で協力することで新たな知見を得やすいという利点はある。

# 知能について(8/11)

- よりよき思考の指向性を得るには？
  - そもそも思考を進める先の選択肢を得ることができなければ指向性も云々以前に立ち往生.
  - その選択肢となりうる情報を見つけ認識することが必要不可欠.
  - よりよき思考の指向性を得る為にはより多くのその選択肢が得られたほうが有利.
  - 洞察力や認識力などといったものが重要.
    - 洞察力→情報を拾う能力
    - 認識力→情報を捉える(扱える)能力

# 知能について(9/11)

- 洞察力および認識力の重要性について
  - どんなに頭がいいとされる人間でもあらゆる情報が遮断された状態ではよりよき判断・答えを導き出すことはできません.
  - 逆に答えそのものに非常に近い情報を得ることができれば、答えに辿り着くのは簡単なことです.

# 知能について(10/11)

- 洞察力および認識力の重要性について
  - 洞察力が低ければ目隠しされているも同然で、逆に洞察力が高ければ答えに近い情報も見つけやすくなる。
  - 認識力が低ければせっかく洞察力によって見つけた情報も意味を成さないし、認識力が高ければ得られた情報を活用しやすくなる。

# 知能について(11/11)

- そして . . .
  - 考えるということは洞察と認識を繰り返すこと.
  - そしてあるべきコンピューティングの姿はこの洞察と認識を支援すること.
    - コンピュータシステム自体も洞察と認識を繰り返す.

Luciferの設計コンセプトと導入予定の機能紹介

# 未来の言語の前に 生物について

# 生物について(1/5)

- 複雑さの克服
  - 生物の体というものはものすごく複雑.
  - しかしとても複雑ではあるがその膨大なエントロピーを前にしてもそのシステムは崩壊していない.

# 生物について(2/5)

## ■ 複雑さの克服

- 生物は如何にして膨大なエントロピーを抱えるシステムを維持しているのか？
- システムが内包するエントロピーに対して少しずつ異なる個体というエントロピーをぶつける。
- 致命的な問題を抱えた個体は自然淘汰により排除。
- 残った個体には大きな問題が存在しない？

# 生物について(3/5)

## ■ 複雑さの克服

- 実際にはさまざまな問題を抱える。
  - 泳げる癖に1時間も水中に潜っていれば溺死。
  - 食道と気道が繋がっている(※)せいで食べ物を喉に詰まらせただけで窒息死。
    - ※一般的な哺乳類はちゃんと分かれている。
  - ウイルスをうっかり細胞内に取り込んで自身のDNAと同様に扱いそのコードを実行してしまう。

# 生物について(4/5)

- 複雑さの克服
  - さまざまな問題を抱えていてもそれを軽減する仕組みを持つ。
    - 痛みや恐怖等により問題から遠ざかる。
      - 苦しみを感じるできない問題には対処できない。
      - 無駄に苦しみを感じて逃げようともしてしまう。
    - 再生能力や免疫系によりシステムを維持する。

# 生物について(5/5)

- 見えることの重要性
  - ▣ 目の誕生というのは生物の進化の歴史の上でも一大イベント。
    - 目の誕生により弱者の淘汰が著しく加速.
    - コンピュータシステム上の問題も的確に見えさえすれば容易く淘汰(修正)されていく.
    - コンピュータシステムの進化を加速する為にも見えるようにすることは非常に重要.

Luciferの設計コンセプトと導入予定の機能紹介

# 未来の言語の前に 未来のコンピューティング

# 未来のコンピューティング(1/7)

- いまのコンピュータはいろいろなナンセンス
  - 例えば何か問題(異常終了する, 誤った挙動する, 異常に動作が重い, 操作が無視される, etc)が起こっても、実際に何が問題なのかその情報をユーザーに的確に伝えられることはまれで、さらにどうすればその問題が解決あるいは回避できるのかを知るのは往々にして困難.
    - 内部が見えないから.
  - ちょっとしたことでもプログラミングが必要になる.
    - 内部を触れないから.
  - コンピュータがなにをやっているのか分からない.
    - そんなことからマルウェアの類が成立してしまう.

# 未来のコンピューティング(2/7)

- もっと可視性があるべき
  - その可視性に見合った操作性も必要.
- オープンソースではなくオープンバイナリ
  - ソースがオープンなだけではあまり意味がない.
  - 稼働中のバイナリが見えて且つ触れる(変更できる)ことが必要.
- ほぼ確実に複雑化は避けられず, 複雑化するということは問題のない完璧なシステムというものは作成できなくなることを意味する.

# 未来のコンピューティング(3/7)

- 未来のビューイングシステム
  - 現在のテキストエディタは数十年もろくに本質的な進化をしていない。
    - 乱暴に言うと扱えるファイルのサイズや画面のサイズが大きくなった程度.
    - 進化らしい進化といえれば入力候補機能(インテリセンスの類)やマクロ機能の類ぐらい.

# 未来のコンピューティング(4/7)

- 未来のビューイングシステム
  - スマートブラケット
    - 現状は括弧の対応付けがわかりにくい。

NOW  
PAINTING

# 未来のコンピューティング(5/7)

- 未来のビューイングシステム
  - 型/クラス,変数/オブジェクト,関数,命令等々に適切なアイコンを付加して表示.
    - 文字より絵のほうが直感的に理解可能.
    - 表示を縮小しても把握しやすい.

NOW  
PAINTING

# 未来のコンピューティング(6/7)

- 未来のビューイングシステム
  - スクロールレスビュー(ナイトメアビュー)
    - 情報を隠さない.
      - 情報を隠されるのは目隠しされるのと同義.
    - 情報をブラウズしやすい.
      - ブラウズできないのでは目隠しされるのと同義.

NOW  
PAINTING

# 未来のコンピューティング(7/7)

- エラーの許容
  - 厳密性との決別
    - 10%あるいは1%以下程度の確立でプリミティブな各種処理が失敗する前提を受け入れるべき。
    - 現在の厳密性を維持していたのではハードウェアの生産コストも高くなり、速度も出ない。
    - 現在そしてこれからのネットワークが強く絡むコンピューティングの世界ではどのみち厳密性の維持には限界もあるしモデルとしてマッチしていない。
  - エラー許容型コンピューティングモデル
    - ネガティブスイッチのような設計の要求。
  - このモデルの利点
    - 自然とフェイルソフトな構造になりやすい。
    - うまくエラーモデルを設計すれば安価で高速なハードウェアを適用可能になる。
    - うまくエラーモデルを設計すれば分散型コンピューティングにも適するモデルになる。
  - 欠点及び疑問点
    - 決定性(再現性)が損なわれる為、問題の再現確認およびデバッグやテストが困難になる可能性がある。
    - いくらエラーを許容すると言ってもあまりにも出鱈目なエラーは許すわけにはいかないだろうし、出鱈目なエラーを出さないようにするには結局の所従来通りの厳密性を必要とする可能性がある。

Luciferの設計コンセプトと導入予定の機能紹介

# 設計コンセプト

# 設計コンセプト (1/4)

- 一般論としてどのような言語を設計すべきか
  - 神性と魔性
    - 宗教的な話じゃなくて数学的な話での神性と魔性.
      - とは言っても宗教も数学も哲学を親とした兄弟みたいなものですが.
    - 神性とは . . .
      - 言い換えるならば直交性の類.
      - 円などの幾何学模様や白紙の美しさは神性に由来.
      - 神性がないと非効率的になる.
      - プログラミング言語で言うなら純粋関数型言語.
    - 魔性とは . . .
      - 言い換えるならば洗練されたエントロピーの類.
      - 自然や動植物の美しさは魔性に由来.
      - 魔性がないと非現実的になる.
      - プログラミング言語で言うならC++やPHP.

# 設計コンセプト (2/4)

- 一般論としてどのような言語を設計すべきか
  - 各種言語等の各種機能・制約が持つ意味の分類まとめ
    - プリミティブな機能
      - チューリングマシン
      - 真性乱数
    - 可視性の向上
      - ピンポイントの情報が見えるだけでは目隠しされてるのと同義.
    - 複雑さの回避
    - 誤りの回避・防止・検出
      - RAII/const/assert
      - 誤りの分類
        - 誤解(misunderstanding)
        - 失敗(failure)
        - 忘却(oblivion)
    - パフォーマンス
    - 挙動ベースではなく意味ベースでの表現・記述

# 設計コンセプト (3/4)

- lucifer としてはどのように設計するのか
  - 目的
    - 統合情報環境 apocalypse とその上で稼働する自立統合情報処理システム flare を作成する為のプログラミング言語.
    - オープンバイナリの実現.

# 設計コンセプト (4/4)

- lucifer としてはどのように設計するのか
  - C++ベース
    - これほど上質なエントロピーを抱えるプログラミング言語は他には存在しない。
    - これほどの洗練されたエントロピーを壺から組み上げるのは並大抵の労力ではなく、これを利用しない手はない。
    - ただ、このエントロピーをうまく取り込むのは非常に難しいのも事実。

Luciferの設計コンセプトと導入予定の機能紹介

# 導入予定の機能紹介

# 導入予定の機能紹介(1/17)

- 豊富な型修飾(plenty type qualifications)
  - 主目的
    - 魔性による「誤りの回避・防止・検出」機能の確保.
  - C++ では `const` が大活躍！
    - 意味が明白になりやすい.
    - 誤りの早期発見にも繋がる.

# 導入予定の機能紹介(2/17)

- 豊富な型修飾(plenty type qualifications)
  - lucifer ではもっと種類を増やす。
    - inaccessible →読み書き不可
      - 参照情報(アドレス)にアクセスさせたくない場合に使用.
    - writable →書き込み可
    - variable →読み書き可
    - readable →読み込み可
    - immutable →不変値
    - anonymous →匿名値(右辺値)
  - ※ volatile についてはどうするか思案中. コンパイル時に不変性が確立されるものとランタイムで不変性が確立されるものについての区別もどうするか思案中.

# 導入予定の機能紹介(3/17)

- 型復元/修飾復元 (type restore / qualification restore)
  - 主目的
    - 複雑さの回避.
  - 型修飾を増やしたらそれぞれのバージョンのオーバーロードメソッドを用意する必要に迫られメソッド数が爆発してしまう.
    - `const` と `volatile` があるだけのC++でも真面目にオーバーロードメソッドを用意してる結構大変.
  - `max(a, b).value = c.value;` のようなことやろうと思うと, `max()` 関数では内容を変更するわけでもないのに引数を writable な状態で受け取らなければならない.
    - 関数の挙動と引数の修飾の不一致.
    - 引数の修飾が関数の挙動を類推するのに役に立たなくなる.
    - 誤解の元.

# 導入予定の機能紹介(4/17)

- 型復元/修飾復元 (type restore / qualification restore)
  - 戻り値の型および修飾を呼び出し元での型および修飾に復元する。
    - 関数の宣言で復元できるように指定した場合だけに機能させる。
    - 復元を実施するのは呼び出し側のコード。
    - 呼び出される側のコードで由来の異なる値を返そうとした場合はコンパイルエラー。
    - \*thisを返すようなメソッドを持つクラスを継承した場合にもそのメソッドを呼ばれた場合に型が継承元のクラスに落ちなくて済むようにできる。

# 導入予定の機能紹介(5/17)

- 完全演算表示 (providence)
  - 主目的
    - 可視性の確保
  - 各関数の取り得る全ての入出力を n 次元グラフ化して表示.
    - できればソースコードのエディット中に.
    - できれば関数に限らず全ての演算の入出力の結果に対して.
  - これが実現できればコードの結果が一目瞭然になり開発効率が向上.
    - バグの発生率も激減する(より正確には発生したそばから発見され即駆逐される).
    - 複雑なロジックでも組みやすくなる.
    - コーディングの完了とテスト・デバッグの完了もほぼ同一のタイミングになる.

# 導入予定の機能紹介(6/17)

- 完全演算表示 (providence)
  - カラーリング例
    - 未計算：無色 or 青
      - 未計算なので出力座標が定まっていない為その入力座標に対する出力軸全体がこの色.
    - 未計算と計算完了の混在：黄
    - 計算完了(正常)：緑
    - 計算完了(異常)：赤
      - assert,throw,OSに異常終了させられるケースなど.
      - 曲がりなりにも値を返すことができる場合はその座標だけがこの色. 値を返せない場合はその入力座標に対する出力軸全体がこの色.

# 導入予定の機能紹介(7/17)

- 完全演算表示 (providence)
  - abs() 関数での例

```
int abs(int a)
{
    return 0 <= a ? a: -a;
}
```



NOW  
PAINTING

- 負の最大値の場合に正の値に変換できずに負の値のまま返してしまっていることが一目瞭然！

# 導入予定の機能紹介(8/17)

- 完全演算表示 (providence)
  - intcmp() での例

```
int intcmp(int a, int b)
{
    return a - b;
}
```



NOW  
PAINTING

- 整数オーバーフローバグの存在が一目瞭然！

# 導入予定の機能紹介(9/17)

- 完全演算表示 (providence)
  - その他の利点
    - 完全な演算が現実的な時間内で完了する場合にはそれを元にコンパイラがソースコードを無視して同一の入出力グラフを描く最適なコードを自動生成可能.
    - 始めからコードではなく入出力グラフでプログラミングを行うという道もある.

# 導入予定の機能紹介(10/17)

- 完全演算表示 (providence)
  - 課題・懸念事項
    - 無尽蔵にマシンパワーを消費する.
    - 文字列や可変配列などの入出力パラメータの扱い.
    - 様々な工夫をこなしても表示に限界があるのは確か.
    - 4次元以上のグラフをどれだけの人間が正しく且つ直感的に  $n$ 次元視できるのか？
    - 4次元以上のグラフを正しく且つ直感的に  $n$ 次元視できる人間であっても高次元の  $n$ 次元視は無理があるだろうし、そもそも表現自体も表現すべき情報を表現できずに意味を成さなくなってくる可能性が高い.

# 導入予定の機能紹介(11/17)

- 超例外処理 (super exception handling)
  - 主目的
    - 処理結果情報と制御の神性及び可視性の確保.
  - 超例外は例外, エラー, assertの類, そして正常系の戻り値をも統合する概念.
  - 木構造を構成.
    - これは今現在の多くのプログラミング言語で欠損している概念.
    - エラー情報の類は木構造を形成できなければ厳密なエラーハンドリングは不可能.
      - 途中で問題が発生しても必ず実行しなければならない後続の処理というものが存在する場合, そこでもまた問題が発生する可能性があり, そのようなエラー情報を扱うには必然的に木構造を形成できることが求められる.
      - そもそも関数呼び出しというものが木構造を形成するのだからその結果(エラー情報)も当然木構造になる.

# 導入予定の機能紹介(12/17)

- 超例外処理 (super exception handling)
  - あるがままを返す。
    - return で指定した以外の途中の演算結果もreturnで指定された値同様に全て呼び出し元へ返す。
    - 正常系/異常系をシームレスに扱う。
      - 何が正常で何が異常なのかは厳密に判断することは難しくまたそもそも根本的な問題として往々にしてその判断は各種処理単位の呼び出される側ではなく呼び出す側の都合に強く依存するのが現実。
      - 契約プログラミングの立場に立てば明確なものとして判断が可能だが、それはあくまで契約を明確にできる場合の話であり、より複雑でよりwetなものになるこれからのコンピューティングでは破綻する。

# 導入予定の機能紹介(13/17)

- 超例外処理 (super exception handling)
  - あるがままを受け取る。
    - 全ての情報を受け取れることで . . .
      - 正常な場合に本当に正しく処理が実行されたか検証可能になる。
      - 異常な場合にどこに問題があったのかすぐに特定できる。

# 導入予定の機能紹介(14/17)

- 超例外処理 (super exception handling)
  - 課題・懸念事項
    - オープンバイナリに基づく機構であり且つオープンバイナリの構成要素でもある為、超例外処理だけでは不完全あり、エンドユーザーにエラー情報が通知された場合にエンドユーザーが自力で問題を解決できる機構が求められる。
      - 超例外はエンドユーザーに問題解決の為に必要な情報を提供する。
    - さすがに現行案を杓子定規にそのまま実装したのではデータ量が現実的な範囲に収まるわけもなく、さまざまな工夫が必要となる。

# 導入予定の機能紹介(15/17)

- 連鎖参照 (presenter)
  - 主目的
    - 値保持型の神性の確保.
  - 参照先を変更できる参照型のようなもの.
  - 参照を自由にネストできる.
    - `int *` と `int **` のような違いが生まれない.

# 導入予定の機能紹介(16/17)

- 連鎖参照 (presenter)
  - 全て連鎖参照型で統一される。
    - `int` と `int*` と `int&` のような違いが生まれない。

```
int a = 0;  
int * a = new int(0);  
Int & a = b;
```



```
int a ← new(stack) int(0);  
int a ← new int(0);  
int a ← &b;
```

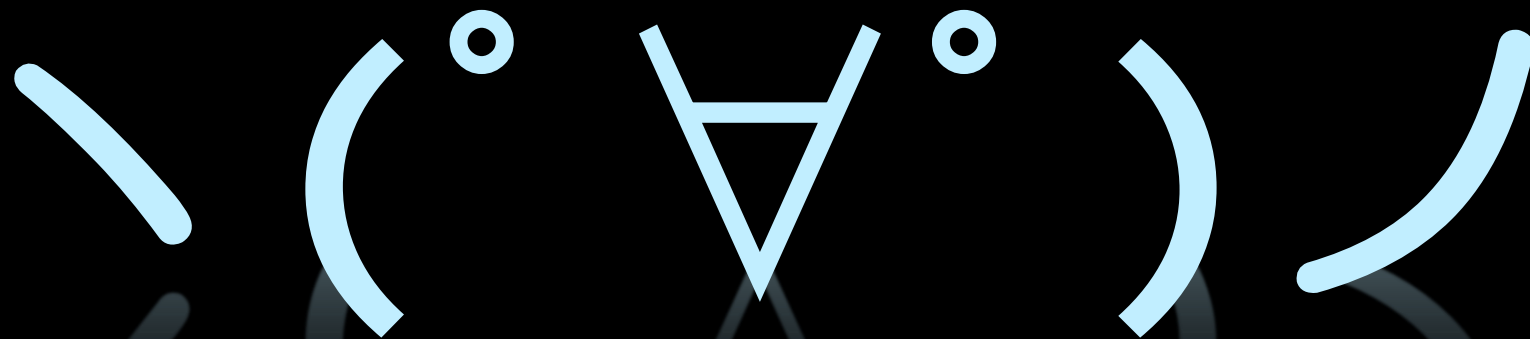
# 導入予定の機能紹介(17/17)

- 連鎖参照 (presenter)
  - 全ての値に対して代入と参照をフックできる。
    - 遅延評価も可能になる。
  - 課題・懸念事項
    - コンパイル時に静的に解決できる場合を除き . . .
      - 値の代入及び参照が関数呼び出しの連鎖となる。
      - 値の代入及び参照のコストが実際に行ってみないと分からない。
      - エラーが発生するリスクも常についてまわる。

Luciferの設計コンセプトと導入予定の機能紹介

# 質疑応答





ご静聴ありがとうございました。