プログラム合成対象言語Pro5LangのPython実装と構文の検討

中田 秀基(順天堂大学)
一杉 裕志、高橋 直人、竹内 泉 (産業技術総合研究所)
佐野 崇(東洋大学)

背景

- Pro5Lang
  - Probabilistic Proven Proposition Processing Programming Language
  - 「確率的証明済み命題処理プログラミング言語」
- AIエージェントが、自らの経験を抽象化した結果として合成する対象の言語として設計
- 脳に対する知見に基づき、脳で実現可能であろうと思われる基礎的な機能のみで構成

- 従来の実装はJavaの言語内DSLとして実装
  - 他の研究者に試用していただくことが困難

目的と成果

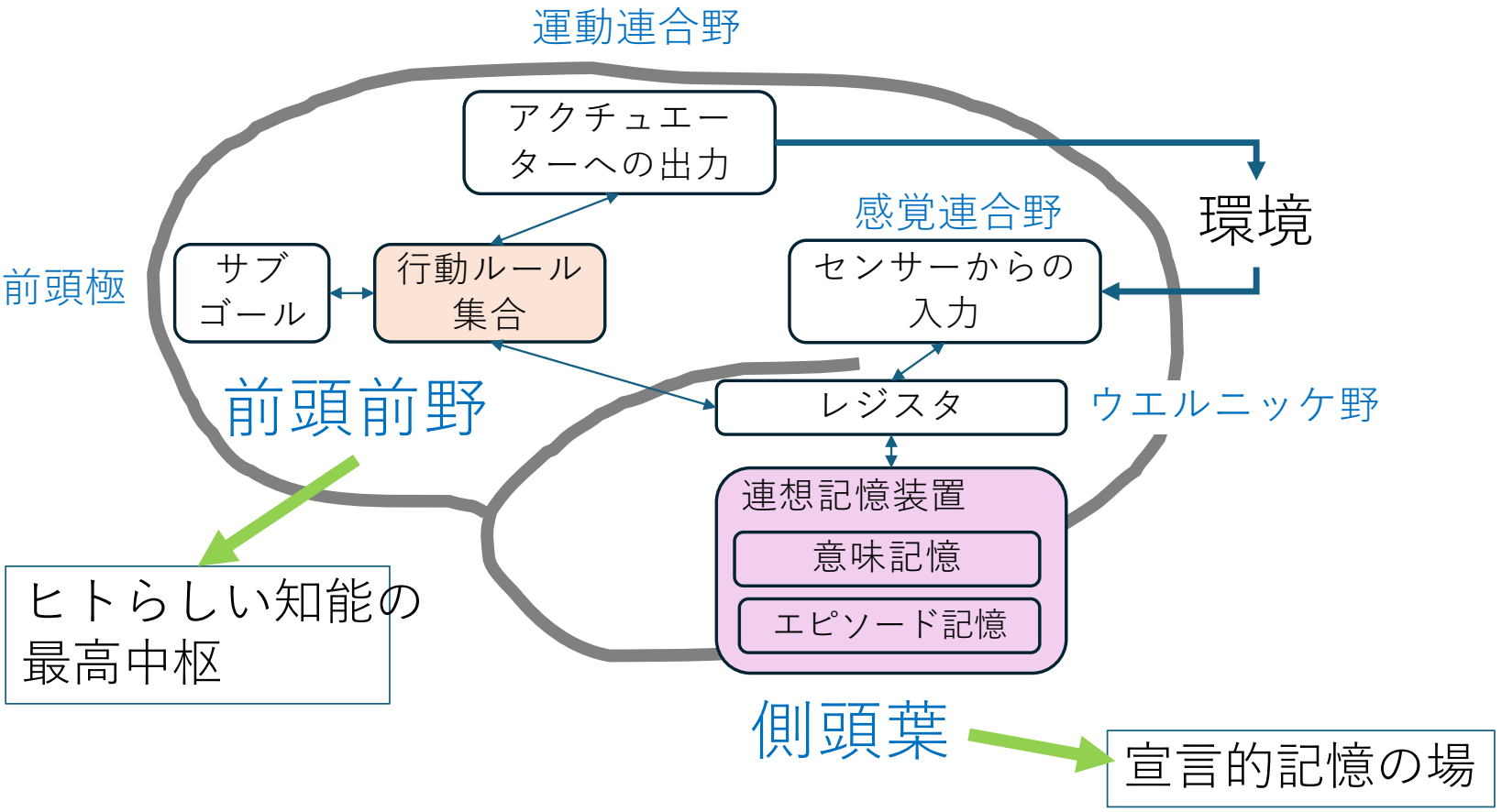
- 他の研究者の試用に供するためのPro5LangのPython実装を提示
  - githubで公開予定

- Pro5Langの新たな構文を提案
  - 他の研究者が使用する際のノイズを低減
  - いくつかの候補を提示

発表の構成

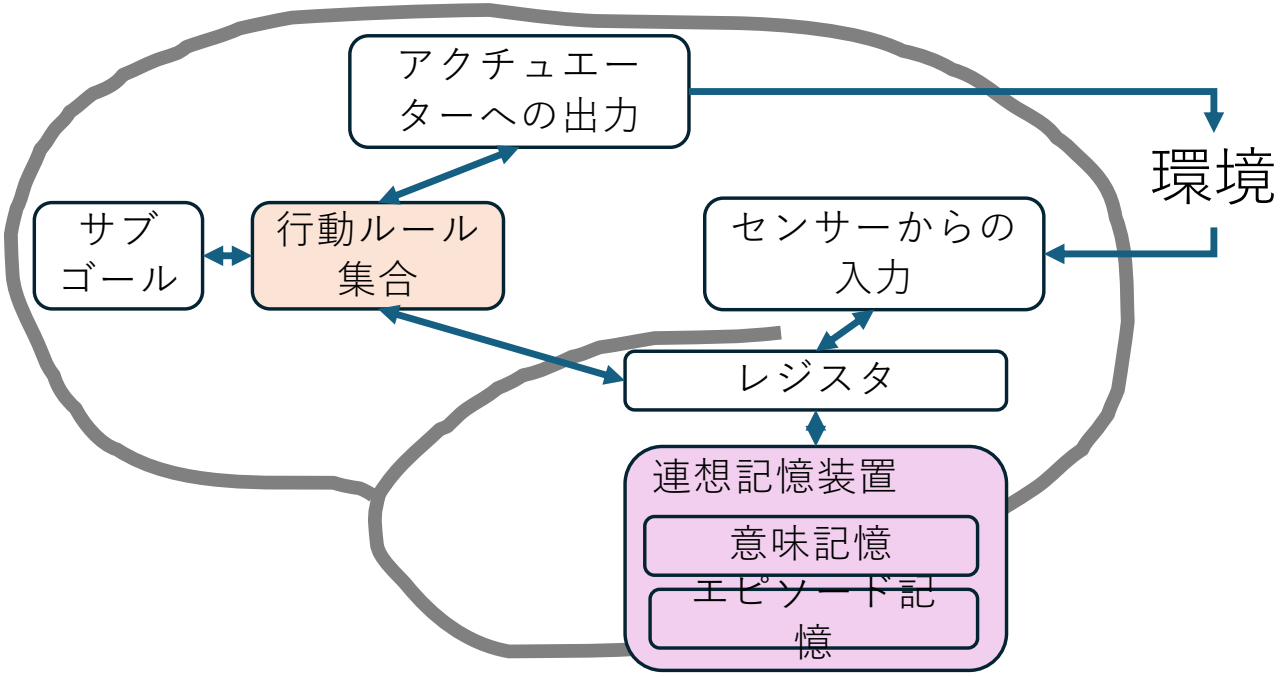
- Pro5Langのコンセプト
- Pro5Langの動作を整理
- Python実装
- 構文に関する検討
- デモ
- おわりに

提案する脳型AGIアーキテクチャの全体像

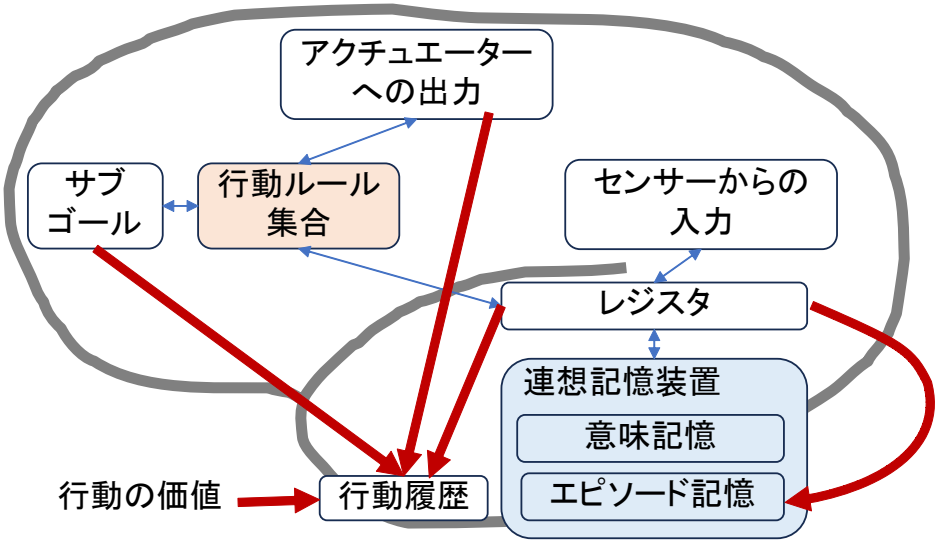


行動ルール集合がアーキテクチャの中心

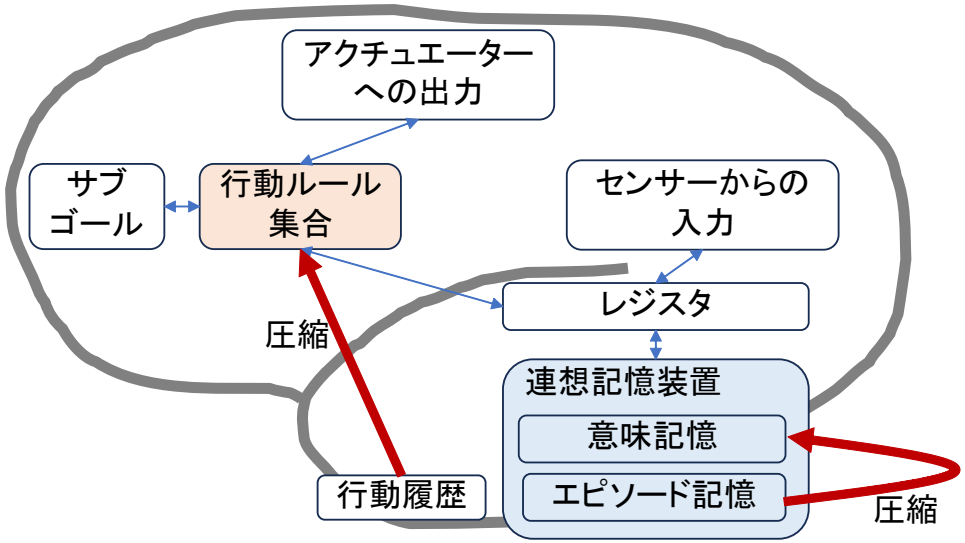
- 行動ルール集合
  ＝ プログラム
  ＝ 行動価値関数 Q(s,g,a)を
  圧縮表現したもの
- 行動、推論、発話、言語理解、
  記憶の想起などの
  「意識的行動」は
  すべて行動ルールに従い実行
- 行動ルールはコンポーネント間の
  ゲートの開閉を制御



知識獲得：ベイジアンネットの学習



行動履歴と認識履歴を記憶



定期的に
行動履歴と認識履歴を圧縮・抽象化し、
手続き的知識と宣言的知識を獲得

合成対象言語 Pro5Lang

- 機械語と論理型言語の特徴を合わせ持った**手続き型言語**
- **機械語に似た特徴**：
  - 固定長のデータ、固定長の命令
  - 少数のレジスタと大容量のメモリ
  - 神経回路での実現が容易
- **論理型言語に似た特徴**：
  - パターンマッチングによる行動ルール選択
  - （学習が理想的に進めば）正しい推論をする
- プログラム（行動ルール集合）は
  教師なし学習と強化学習で獲得

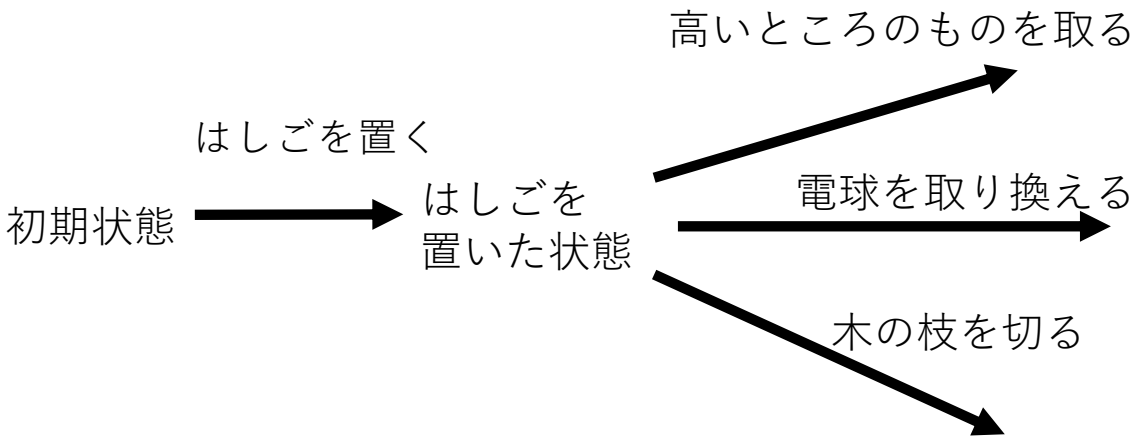
```
 1: k(e(0, 0, Socrates, Is, AMan, 0));       // ソクラテスは人間である
 2: k(e(If, c(0, 0, x, Is, AMan, 0), c(0, 0, x, Is, Mortal, 0))); // 人間は死ぬ

 3: eifxy = e(If, c(x1,x2,x3,x4,x5,x6), c(y1,y2,y3,y4,y5,y6));
 4: eif0y = e(If, c(__,__,__,__,__,__), c(y1,y2,y3,y4,y5,y6));
 5: ax = a(x1,x2,x3,x4,x5,x6);
 6: ay = a(y1,y2,y3,y4,y5,y6);
 7: rule(w(),       w(ay), recall(eif0y));  // y を証明するために x → y を想起
 8: rule(w(eifxy),  w(ay), call(ax));       // x → y なら y を証明するために x を証明
 9: rule(w(ax),     w(ay), recall(eifxy));  // x なら y を証明するために x → y を想起
10: rule(w(ax, eifxy), w(ay), set(ay));     // x, x → y なら y が成り立つ

11: ex = e(x1,x2,x3,x4,x5,x6);
12: rule(w(),   w(ax), recall(ex));
13: rule(w(ex), w(ax), set(ax));
```

プロトタイプ版 Pro5Lang のプログラム例

Pro5Lang と強化学習

- 行動価値関数 Q(s,g,a) を圧縮抽象化したものが
  Pro5Lang プログラム ＝ 行動ルール集合
- 価値の高い行動ルールを選択し実行 → 報酬最大化
- サブルーチンを再帰的に呼び出すことが可能
  - 再帰的強化学習アルゴリズム RGoal [一杉+ 第26回 汎用人工知能研究会, 2024]
- サブルーチンの価値の定義： MAXQ 価値観数分解[Dietterich 2000]の拡張
- **推論規則の「正しさ」を「価値」で代用**[一杉+ 第14回 汎用人工知能研究会, 2020]
  → 経験から「正しい知識」を獲得可能に

サブルーチンの例



最終目的は違っていても、
「はしごを置く」というサブルーチンは共通

サブタスク共有

「はしごを置く」サブルーチンを実行中は
はしごの周辺以外の情報をすべて無視

状態抽象

Pro5Langの動作

- 内部状態を、ルールを用いて更新する
  - 現在のゴールと内部状態にマッチするルールを選択
  - 内部状態が、ゴールを満たすようになったらゴール達成

- ゴールを達成するためにサブゴールを呼び出すことができる
  - 現在のゴールは「ゴールスタック」にpushされる
  - サブゴールが達成されたらゴールスタックからゴールをpopして続行

- 適用できるルールがなくなったら、ゴールスタックの底にあるゴールを取り出して、再実行
  - 場合によっては無限ループする

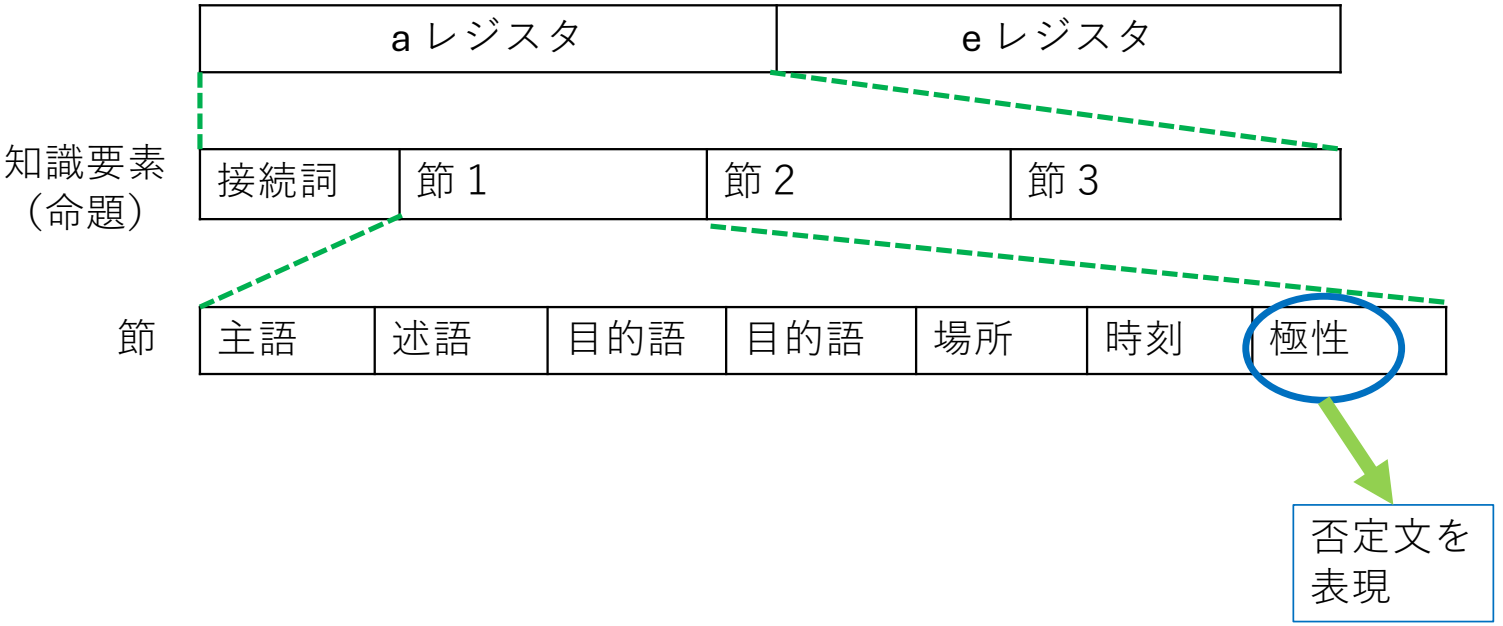
実行モデル:

- 2つのレジスタ
  - Aレジスタ
    - 直接設定することができる。
    - このレジスタの値がサブゴールの条件を満たすことが動作の目的
  - Eレジスタ
    - 直接設定することはできず、記憶から「想起」することだけができる。
- 記憶
  - 想起の対象となる
  - 宣言的記憶:プログラムの一部として与えられる
  - エピソード記憶: プログラムの実行過程でセットされる
    - Aレジスタにsetを行うと、自動的に記憶に挿入される

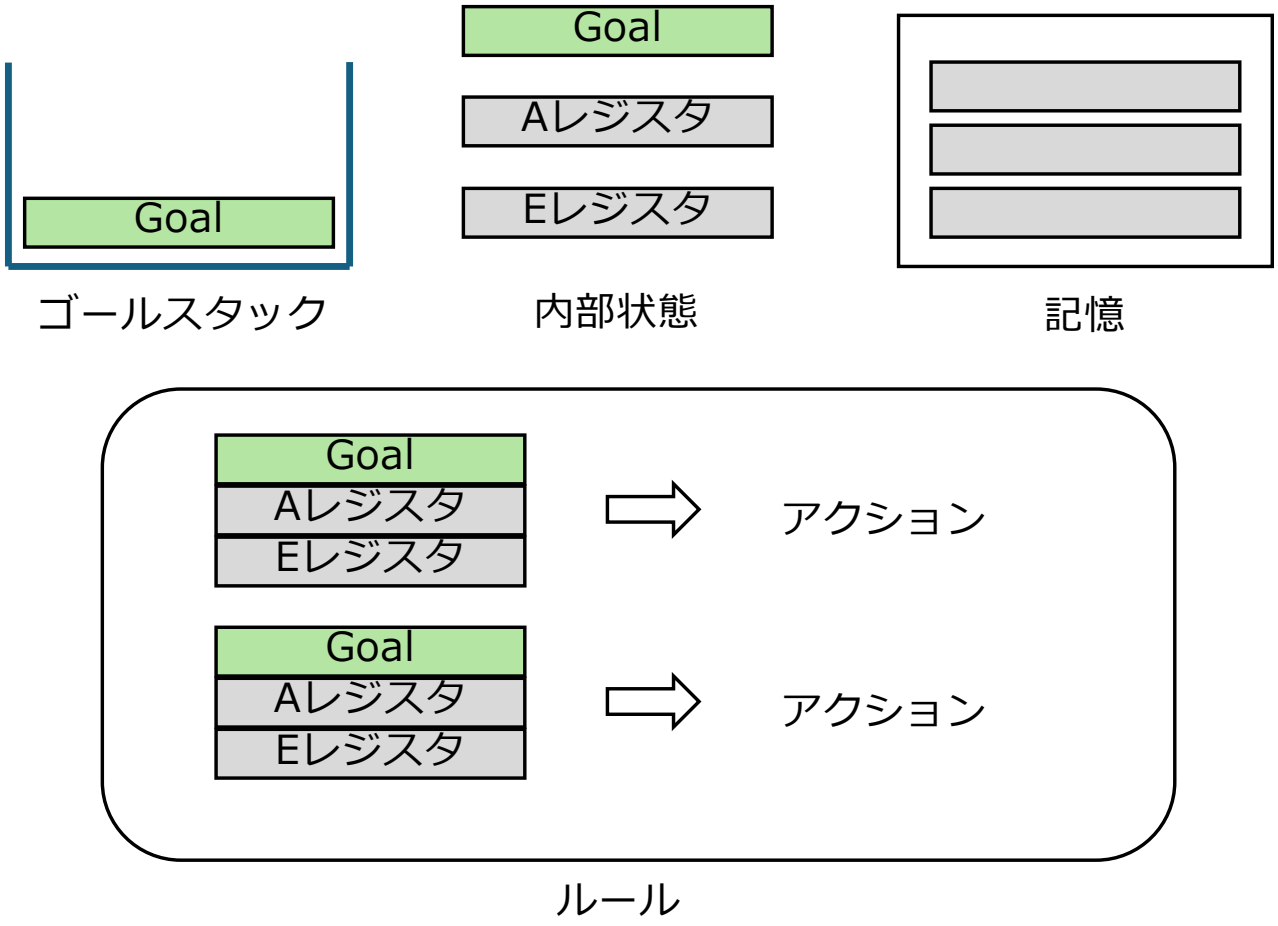
Pro5Langのプログラム

- プログラムの目的は、「ゴール」で提示される内部状態を実現することである
  - 具体的にはレジスタの1つが「ゴール」の条件を満たす
- プログラムは(複数の)ルールで構成される
  - ルール
    - 現在の「状態」にマッチするための「パターン」
    - マッチした場合に状態を書き換える「アクション」
- 状態
  - 2つの「レジスタ」 – ルールでマッチできるのはこれだけ
  - 「記憶」はルールの対象にはならない

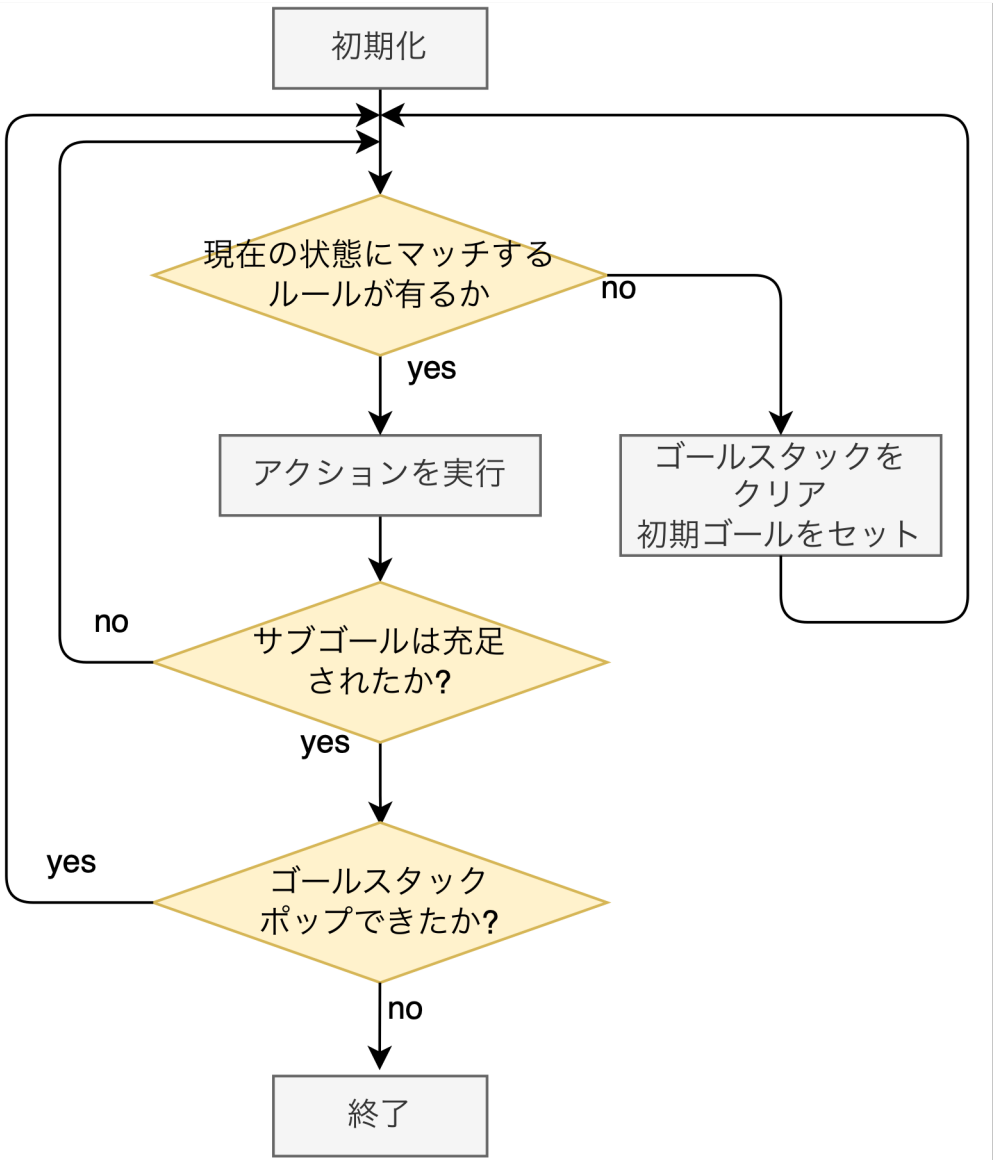
データ構造



Pro5Langの構成要素



フローチャート

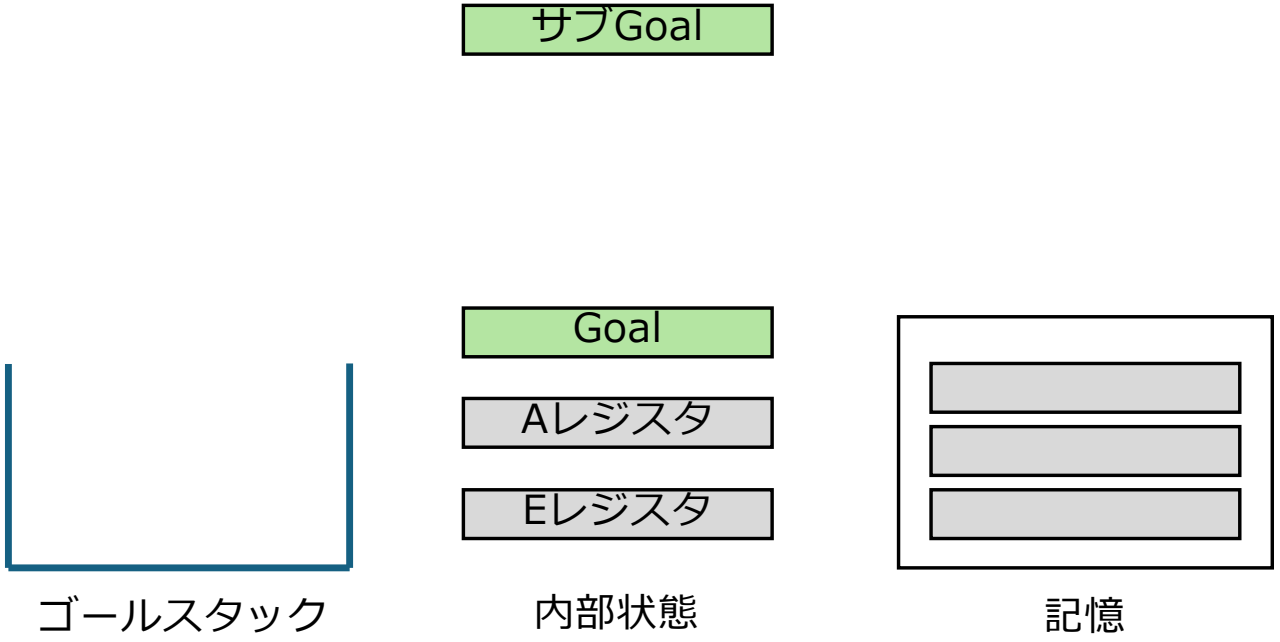

初期化
現在の状態にマッチする
ルールが有るか
no
yes
アクションを実行
ゴールスタックを
クリア
初期ゴールをセット
no
サブゴールは充足
されたか?
yes
yes
ゴールスタック
ポップできたか?
no
終了

アクション

- 「状態」を更新する
- call
  - サブゴールを呼び出す
  - 現在のゴールをゴールスタックにプッシュし、新たなサブゴールをゴールとしてセット
  - 再帰的なタスク分割に用いる
- set
  - Aレジスタへ値をセット
  - 同時に記憶に保存
- recall
  - 記憶からパターンマッチで想起
  - 想起した内容をEレジスタに保存

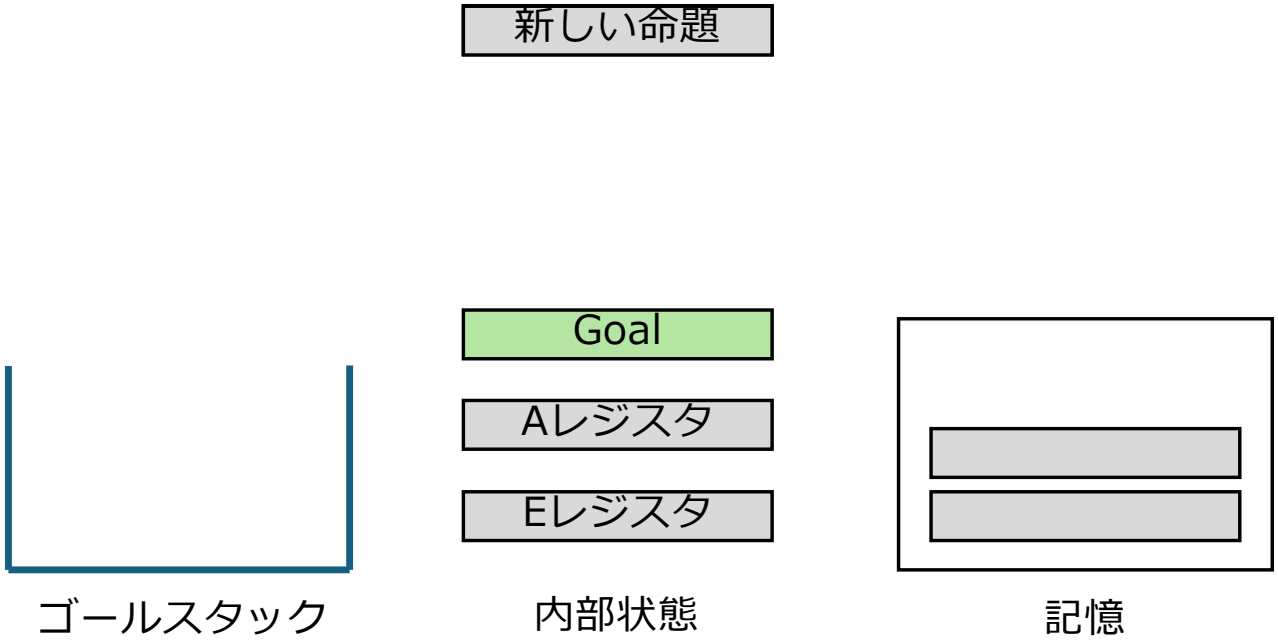
Callの動作

- 現在のゴールをゴールスタックにプッシュし、新たなサブゴールをゴールとしてセット



set

- Aレジスタへ値をセット
- 同時に記憶に保存



recall

- 記憶からパターンマッチで想起
- 想起した内容をEレジスタに保存

Pro5Langの動作例: 叔父、甥の発見

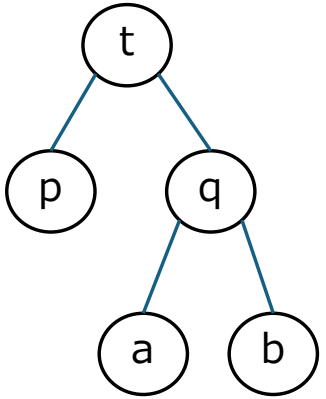
親(t, p), 親(t, q), 親(q, a), 親(q, b)

兄弟(_,_), _ , _ => recall(親(+,+))

兄弟(_,_), _, 親(X,Y) => set(親(X, Y))

兄弟(_,_), 親(X,Y), _ => recall(親(X, +))

兄弟(_,_), 親(X,Y), 親(X,Z) => set(兄弟(Y, Z))

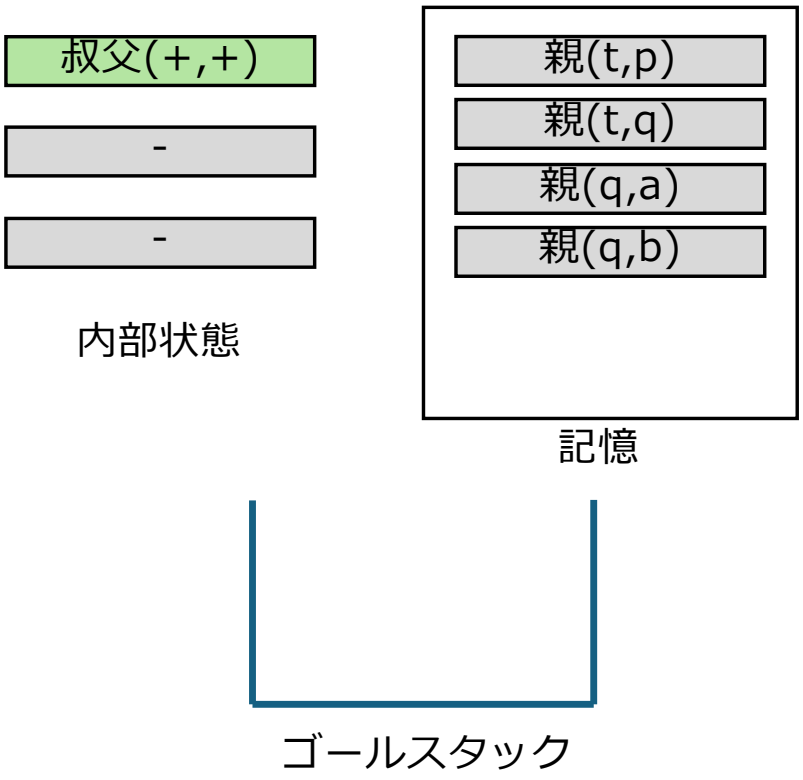


叔父(_,_), _ ,_ => call(兄弟(+,+))

叔父(_,_), 兄弟(X,Y),_ => recall(親(Y, +))

叔父(_,_), 兄弟(X,Y),親(Y,Z)=> set(叔父(X, Z))

動作(1)



親(t, p), 親(t, q), 親(q, a), 親(q, b)

兄弟(_,_), _ , _ => recall(親(+,+))

兄弟(_,_), _, 親(X,Y) => set(親(X, Y))

兄弟(_,_), 親(X,Y), _ => recall(親(X, +))

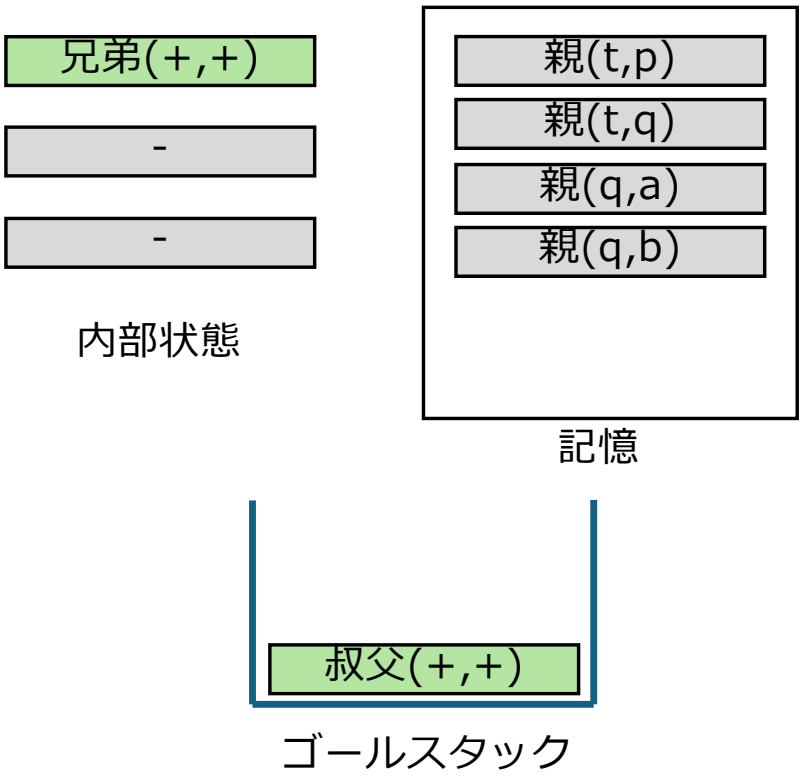
兄弟(_,_), 親(X,Y), 親(X,Z) => set(兄弟(Y, Z))

叔父(_,_), _ ,_ => call(兄弟(+,+))

叔父(_,_), 兄弟(X,Y),_ => recall(親(Y, +))

叔父(_,_), 兄弟(X,Y),親(Y,Z)=> set(叔父(X, Z)

動作(2)



親(t, p), 親(t, q), 親(q, a), 親(q, b)

兄弟(_,_), _ , _ => recall(親(+,+))

兄弟(_,_), _, 親(X,Y) => set(親(X, Y))

兄弟(_,_), 親(X,Y), _ => recall(親(X, +))

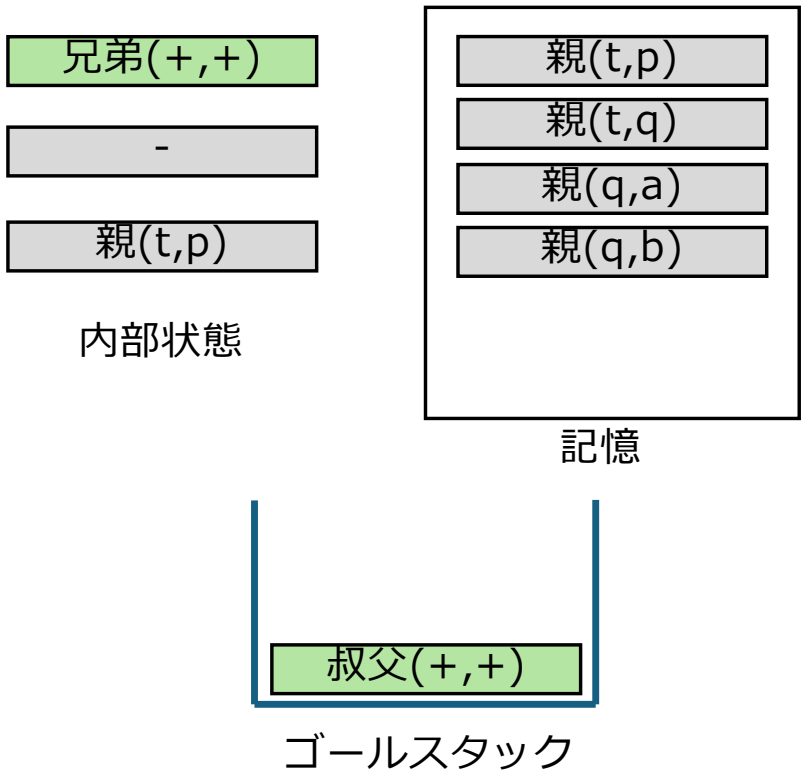
兄弟(_,_), 親(X,Y), 親(X,Z) => set(兄弟(Y, Z))

叔父(_,_), _ ,_ => call(兄弟(+,+))

叔父(_,_), 兄弟(X,Y),_ => recall(親(Y, +))

叔父(_,_), 兄弟(X,Y),親(Y,Z)=> set(叔父(X, Z))

動作(3)



親(t, p), 親(t, q), 親(q, a), 親(q, b)

兄弟(_,_), _, _ => recall(親(+,+))

兄弟(_,_), _, 親(X,Y) => set(親(X, Y))

兄弟(_,_), 親(X,Y), _ => recall(親(X, +))

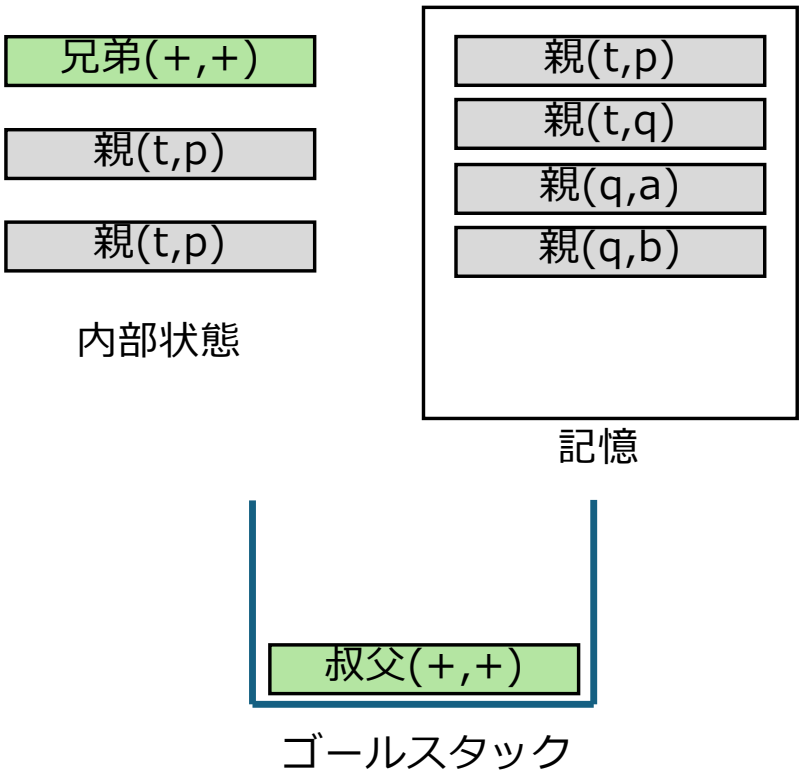
兄弟(_,_), 親(X,Y), 親(X,Z) => set(兄弟(Y, Z))

叔父(_,_), _, _ => call(兄弟(+,+))

叔父(_,_), 兄弟(X,Y),_ => recall(親(Y, +))

叔父(_,_), 兄弟(X,Y),親(Y,Z)=> set(叔父(X, Z))

動作(4)



親(t, p), 親(t, q), 親(q, a), 親(q, b)

兄弟(_,_), _ , _ => recall(親(+,+))

兄弟(_ , _), _ , 親(X,Y) => set(親(X, Y))

兄弟(_,_), 親(X,Y), _ => recall(親(X, +))

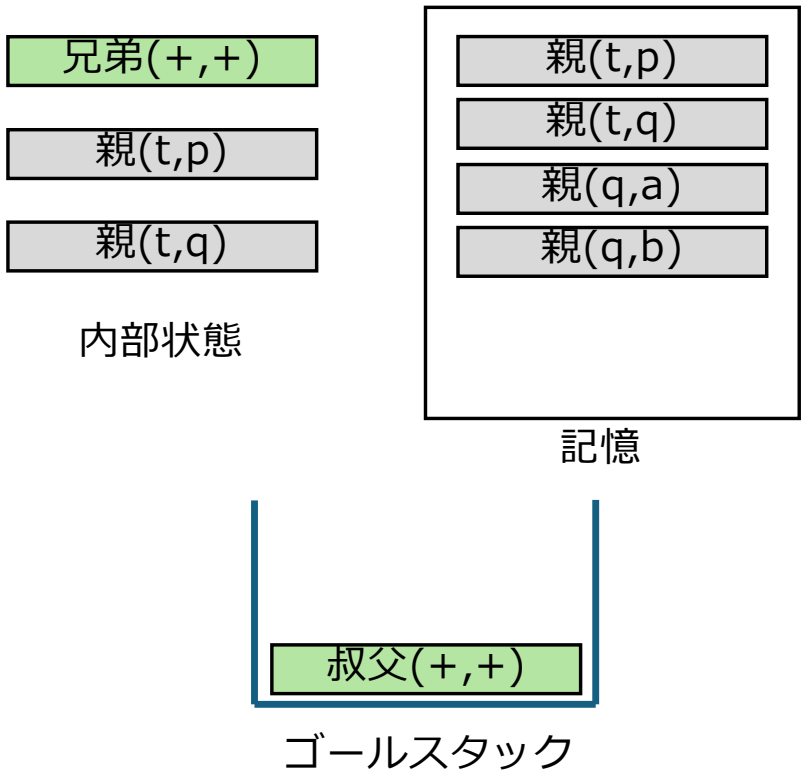
兄弟(_,_), 親(X,Y), 親(X,Z) => set(兄弟(Y, Z))

叔父(_,_), _ ,_ => call(兄弟(+,+))

叔父(_,_), 兄弟(X,Y),_ => recall(親(Y, +))

叔父(_,_), 兄弟(X,Y),親(Y,Z)=> set(叔父(X, Z))

動作(5)



親(t, p), 親(t, q), 親(q, a), 親(q, b)

兄弟(_,_), _ , _ => recall(親(+,+))

兄弟(_,_), _, 親(X,Y) => set(親(X, Y))

兄弟(_,_), 親(X,Y), => recall(親(X, +))

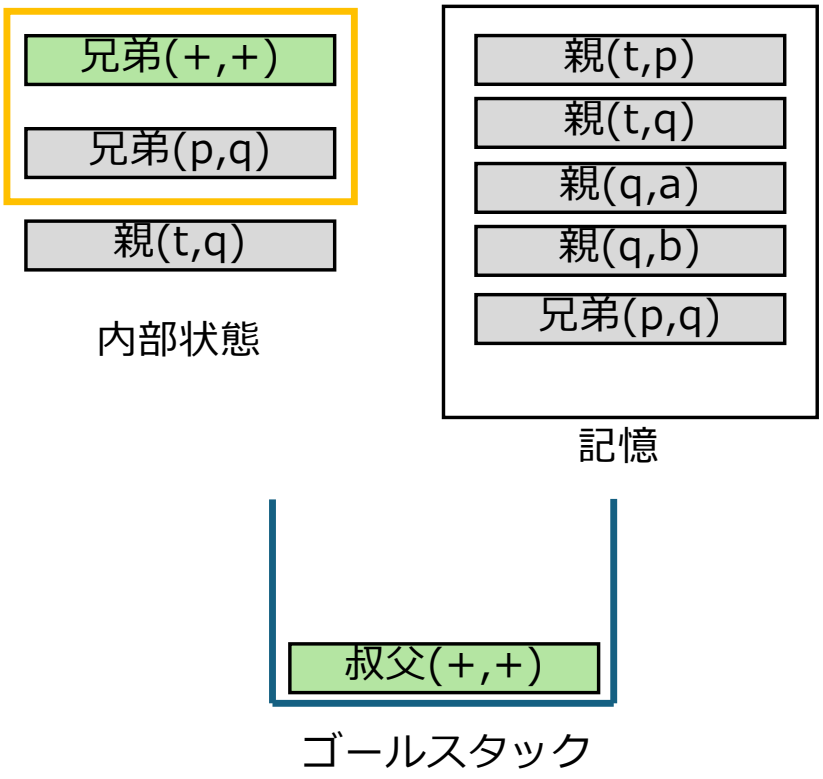
兄弟(_,_), 親(X,Y), 親(X,Z) => set(兄弟(Y, Z))

叔父(_,_), _ ,_ => call(兄弟(+,+))

叔父(_,_), 兄弟(X,Y),_ => recall(親(Y, +))

叔父(_,_), 兄弟(X,Y),親(Y,Z)=> set(叔父(X, Z))

動作(6)：サブゴール達成



親(t, p), 親(t, q), 親(q, a), 親(q, b)

兄弟(_,_), _ , _ => recall(親(+,+))

兄弟(_,_), _, 親(X,Y) => set(親(X, Y))

兄弟(_,_), 親(X,Y), _ => recall(親(X, +))

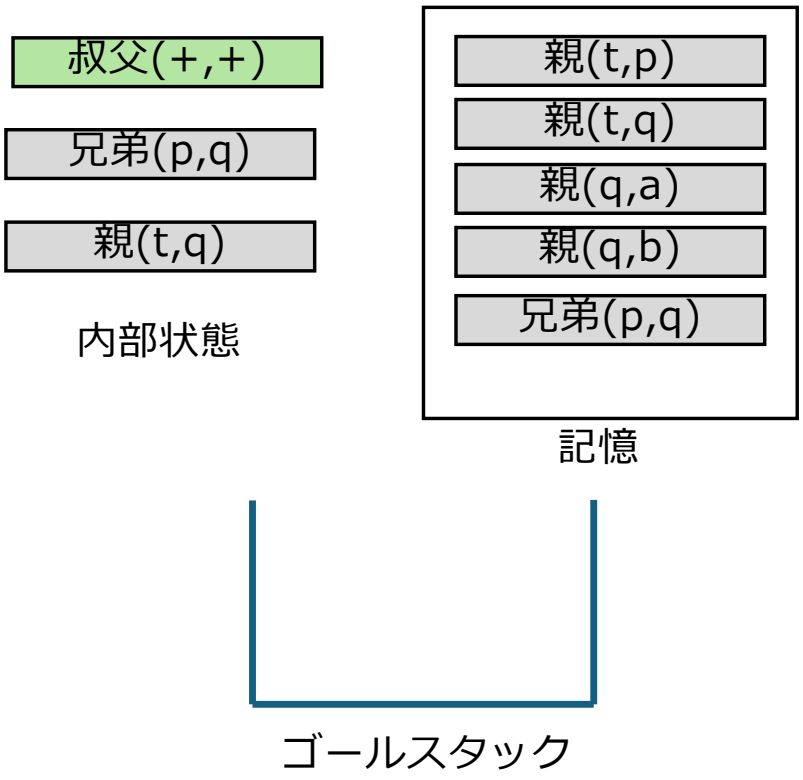
兄弟(_,_), 親(X,Y), 親(X,Z) => set(兄弟(Y, Z))

叔父(_,_), _ ,_ => call(兄弟(+,+))

叔父(_,_), 兄弟(X,Y),_ => recall(親(Y, +))

叔父(_,_), 兄弟(X,Y),親(Y,Z)=> set(叔父(X, Z))

動作(7)：



親(t, p), 親(t, q), 親(q, a), 親(q, b)

兄弟(_,_), _ , _ => recall(親(+,+))

兄弟(_,_), _, 親(X,Y) => set(親(X, Y))

兄弟(_,_), 親(X,Y), _ => recall(親(X, +))

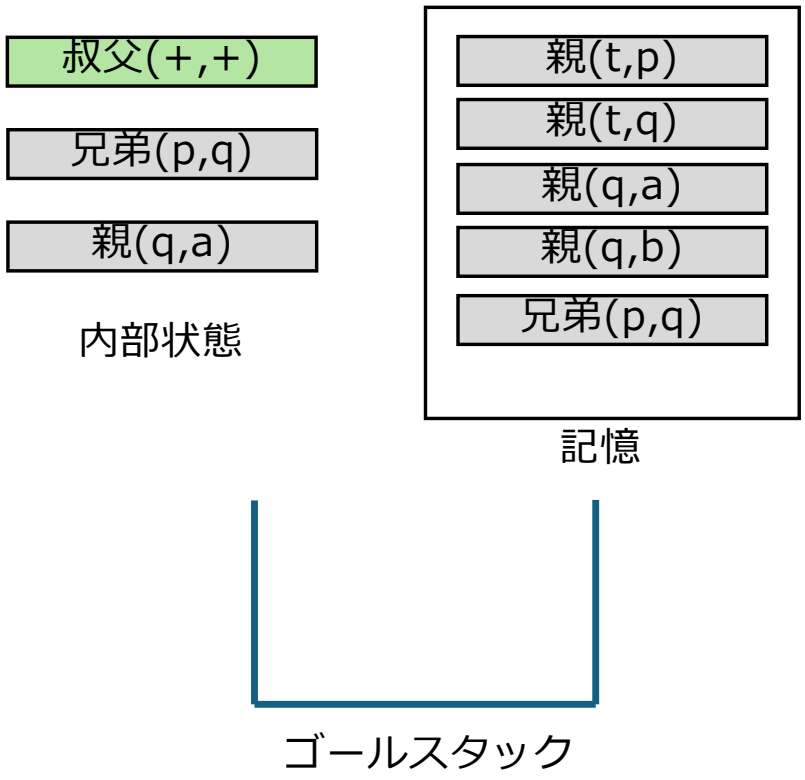
兄弟(_,_), 親(X,Y), 親(X,Z) => set(兄弟(Y, Z))

叔父(_,_), _ , _ => call(兄弟(+,+))

叔父(_,_), 兄弟(X,Y),_ => recall(親(Y, +))

叔父(_,_), 兄弟(X,Y),親(Y,Z)=> set(叔父(X, Z))

動作(8)：



親(t, p), 親(t, q), 親(q, a), 親(q, b)

兄弟(_,_), _ , _ => recall(親(+,+))

兄弟(_,_), _, 親(X,Y) => set(親(X, Y))

兄弟(_,_), 親(X,Y), _ => recall(親(X, +))

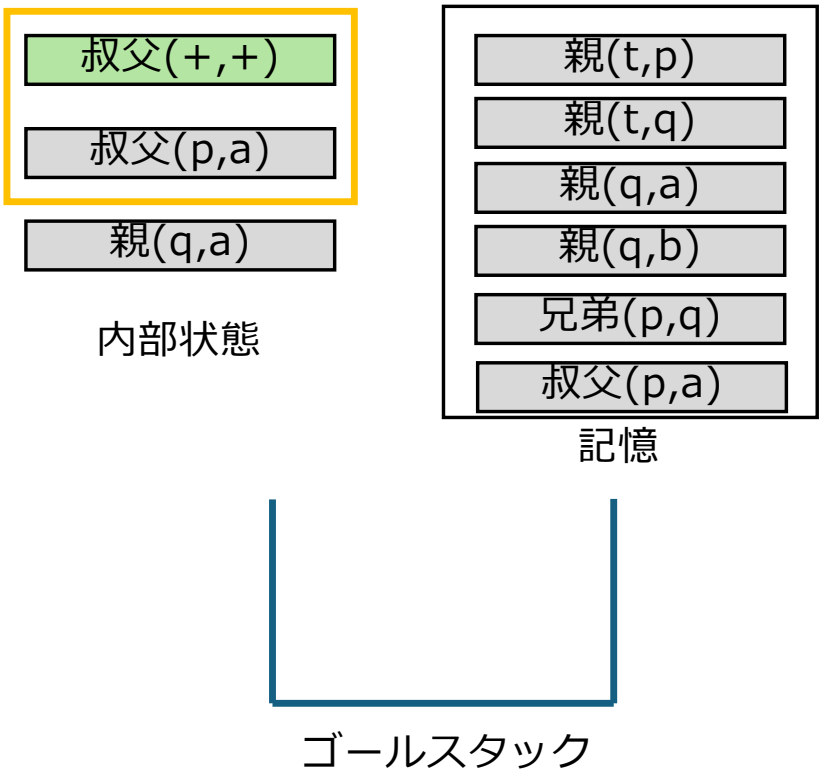
兄弟(_,_), 親(X,Y), 親(X,Z) => set(兄弟(Y, Z))

叔父(_,_), _ ,_ => call(兄弟(+,+))

叔父(,), 兄弟(X,Y), => recall(親(Y, +))

叔父(_,_), 兄弟(X,Y),親(Y,Z)=> set(叔父(X, Z))

動作(9)：ゴール達成



親(t, p), 親(t, q), 親(q, a), 親(q, b)

兄弟(_,_), _ , _ => recall(親(+,+))

兄弟(_,_), _, 親(X,Y) => set(親(X, Y))

兄弟(_,_), 親(X,Y), _ => recall(親(X, +))

兄弟(_,_), 親(X,Y), 親(X,Z) => set(兄弟(Y, Z))

叔父(_,_), _ ,_ => call(兄弟(+,+))

叔父(_,_), 兄弟(X,Y),_ => recall(親(Y, +))

叔父(_,_), 兄弟(X,Y),親(Y,Z)=> set(叔父(X, Z))

既存処理系の問題点

- 基本的に機能検証用で第三者の使用を前提としていない
  - Pro5Lang言語単体として提供できる状況にない
- Javaの言語内DSLとして実装されている
  - Java言語という時点で敬遠されがち
  - 言語内DSL(Domain Specific Language)
    - ホスト言語(Java)の構文を利用して対象言語を表現
      - Javaは言語内DSLに向いている言語ではない
      - c.f. Ruby, Rust

PythonとJavaの比較

- 共通点
  - GCあり
  - オブジェクト指向言語

- Java
  - 静的型付け
    - コンパイル時に型チェックが行われるため頑健
  - 比較的高速
  - 演算子オーバロードなし
  - Oracle買収以降モメンタムを失う

- Python
  - 動的型付け
    - 型アノテーションは書ける
    - 静的チェックはヒント程度
  - 遅い(最近は多少速くなった?)
  - 演算子オーバーロード可能
  - AI分野で圧倒的な利用者数

Python実装

- ポイント
  - 静的型付けがないことによって失われるデメリットを可能な限り解消
  - 頑健性
  - 保守性

- 型アノテーションを可能な限り付与
  - pylanceによる事前チェックで型不整合を検出

- ユニットテストを導入
  - unittestを使用

パーサの実装

- PEGベースのパーサコンビネータライブラリ pyparsingを使用
- パーサコンビネータとは
  - 言語の各構成要素をパースする関数をボトムアップに組み合わせることで、より高位の構成要素のパーサを構築する手法
  - 演算子オーバロードが可能な言語では、BNFに近い記法でパーサを記述できる

PEG (Parsing Expression Grammars)

- 文法クラスの一つ(LL(1)などと同等)
  - バックトラックを用いた動作
  - 最悪計算時間が膨大である可能性がある
    - Packrat parserの導入により実用上は問題ない
    - 最近では広く使われている
      - Python自体のパーサなど
- lexical analyzer(lexなど)とgrammar compiler(yacc/bisonなど)を分割して考える必要がない
  - 終端記号の定義と構文定義を1つのフレームワークで簡潔に記述

パーサのコード(抜粋)

- BNFに近い構文で容易に記述できる

```
atom = pp.Word(pp.srange("[a-z]"), pp.srange("[a-zA-Z0-9_]")).\
          set_parse_action(lambda x: Atom(x[0]))
namedvar = pp.Word(pp.srange("[A-Z]"), pp.srange("[a-zA-Z0-9_]")).\
          set_parse_action(lambda x: NamedVar(x[0]))

term = (slot + LPAREN + slot + COMMA + slot + RPAREN)\
| (slot + LPAREN + slot + COMMA + slot + COMMA + slot + RPAREN)\
| (slot + LPAREN + slot + COMMA + slot + COMMA + slot + COMMA + slot + RPAREN)\
| (slot + LPAREN + slot + COMMA + slot + COMMA + slot + COMMA + slot + COMMA + slot + RPAREN)\
| (slot + LPAREN + slot + COMMA + slot + COMMA + slot + COMMA + slot + COMMA + slot + COMMA + boolean + RPAREN)
```

言語構文の検討

- Pro5Langのプログラムは学習の結果生成されることを想定
  - したがって構文はそれほど重要ではない(機械が書いて機械が読む)
  - とはいえ、人間が検証することも現時点では重要
- 既存実装の問題点
  - Javaの言語内DSLとして実装したことによる制約
    - 余分な構造が必要
    - 特殊なキャラクタが使用できない
- 言語構文への要請
  - 既存言語との類似性
    - 動作機構の類似した言語と構文的に似せることで、認知負荷を下げる
  - 動作機構との直感的一致
    - 動作機構を誤解させることのないようにする

既存言語内DSL実装の問題点

- すべての識別子をどこかで定義する必要がある
- k, e, c, w, rule など、ノイズになる構成要素が存在

```
 1: k(e(0, 0, Socrates, Is, AMan, 0));       // ソクラテスは人間である
 2: k(e(If, c(0, 0, x, Is, AMan, 0), c(0, 0, x, Is, Mortal, 0))); // 人間は死ぬ

 3: eifxy = e(If, c(x1,x2,x3,x4,x5,x6), c(y1,y2,y3,y4,y5,y6));
 4: eif0y = e(If, c(__,__,__,__,__,__), c(y1,y2,y3,y4,y5,y6));
 5: ax = a(x1,x2,x3,x4,x5,x6);
 6: ay = a(y1,y2,y3,y4,y5,y6);
 7: rule(w(),       w(ay), recall(eif0y));  // y を証明するために x → y を想起
 8: rule(w(eifxy),  w(ay), call(ax));       // x → y なら y を証明するために x を証明
 9: rule(w(ax),     w(ay), recall(eifxy));  // x なら y を証明するために x → y を想起
10: rule(w(ax, eifxy), w(ay), set(ay));     // x, x → y なら y が成り立つ

11: ex = e(x1,x2,x3,x4,x5,x6);
12: rule(w(),   w(ax), recall(ex));
13: rule(w(ex), w(ax), set(ax));
```

構文候補案

- ルールの構成
  - ゴールパターン、レジスタパターンx2、アクション
  - 多くの場合、同一のゴールパターンに対して複数のルールが存在する

- パターンマッチを自然に表現したい
  - パターンマッチを構文要素として持つ言語から構文を借用する

- 論理型言語、関数型言語
  - パターンマッチが言語機能の根幹であるため相性がいい
  - Prolog, Haskel

- パターンマッチを持つ通常の言語
  - Rustなど

論理型言語系候補

- Prolog
  - Prologのプログラムはホーン節と呼ばれる構造を記述する。
  - ホーン節は1つのヘッド部と(0個からn個の)ボディ部で構成される。
  - ゴールがヘッド部にマッチするルールが選択される
  - ヘッド部とボディ部の区切りは :- と書く

```
ヘッド部:- ボディ1, ボディ2... .
```

  - サンプル: リストの接続

```
append([A|B], C, [A,D]):- append(B, C, D).
append([], X, X).
```

flat GHC/KL1

- flat GHC / KL1
  - 第5世代コンピュータプロジェクトの言語
  - GHC- Guarded Horn Clauses
    - ホーン節に「ガード」を追加したもの

```
ヘッド部:- ガード条件1,条件2... | ボディ1, ボディ2... .
```

LMNtal(Ueda,05)

- グラフ書換え言語
  - 論理変数を用いてグラフを表現し、それをルールで書き換える
- Prologのホーン節を拡張し、ヘッド部に複数の項を書く
  - 複数の項に同時にマッチして、ボディ部を展開する

```
ヘッド1,ヘッド2... :- ボディ1, ボディ2... .
```

- プログラム例:
  - リストのアペンド

```
append(X0,Y,Z0), c(A,X,X0) :- c(A,Z,Z0), append(X,Y,Z)
append(X0,Y,Z0), n(X0) :- Y=Z0
```

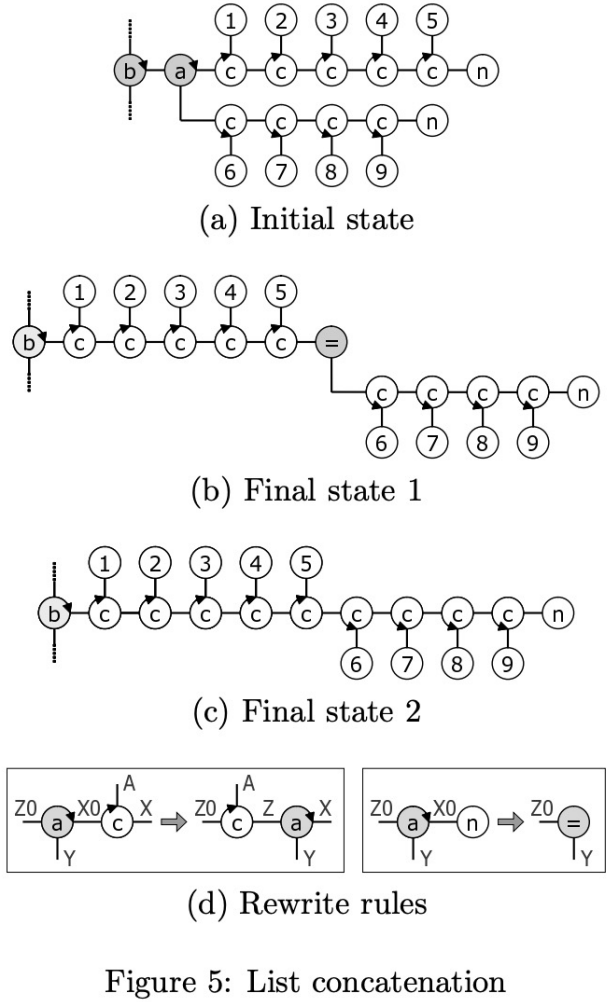


Figure 5: List concatenation

LMNtal: A Language Model with Links and Membranes (WMC2004) より

LMNtalインスパイアドのプログラム例

```
exists(chocolate, -, -, home, yesterday).
exists(snack, -, -, home, yesterday).
notEat(brother, chocolate, home, today).


exists(+, -, -, home, today), _, _ :-
              recall(exists(+, -, -, home, yesterday)).
exists(+, -, -, home, today),
       _,
       exists(X, -, -, home, yesterday) :-
              set(exists(X, -, -, home, yesterday)).

exists(+, -, -, home, today),
       exists(X, -, -, home, yesterday) :-
              recall(notEat(brother, X, -, home, today)).

exists(+, -, -, home, today),
       exists(X, -, -, home, yesterday),            // きのう x があり、
       notEat(brother, X, -, home, today) :-        // 兄が x を食べていないならば、
              set(exists(X, -, -, home, today)).    // x がある
```

Haskell

- 関数型言語
  - 関数の引数の型で複数の定義をディスパッチする

```
append :: [a] -> [a] -> [a]

append [] ys = ys
append (x:xs) ys = x : append xs ys
```

- 関数名をゴールパターン、引数部分をレジスタパターンに置き換える事が考えられる

Rust

- 代数的データ型を持つ
  - 型AもしくはBのような
- 代数的データ型に対する処理をmatch文で行う
  - switch-case文のようなものだが遥かに強力

```
match x {
    None => None,
    Some(i) => Some(i + 1),
}
```

Rustインスパイアドのプログラム例

```
exists(chocolate, -, -, home, yesterday).
exists(snack, -, -, home, yesterday).
notEat(brother, chocolate, home, today).

exists(+, -, -, home, today) {
       _’ _
              => recall(exists(+, -, -, home, yesterday)), # きのうなにがあったかを想起
       _, exists(X, -, -, home, yesterday)
              => set(exists(X, -, -, home, yesterday)), # Eレジスタから Aレジスタにコピー

       exists(X, -, -, home, yesterday), _
              => recall(notEat(brother, X, -, home, today)), #今日兄が食べたかどうかを想起

       exists(X, -, -, home, yesterday), # きのう x があり、
       notEat(brother, X, -, home, today) # 今日兄がxを食べていなければ
              => set(exists(X, -, -, home, today)), # 今日家にxがある
}
```

現時点の構文検討

- アトムと変数はProlog流
  - 小文字から始まる識別子はアトム
  - 大文字から始まる識別子は変数

- とりあえずLMNtal流の記述を採用

ゴールP、AレジスタP、EレジスタP :- アクション .

- パーサの書き換えは容易
  - 実際に実装してさまざまなスタイルを検討する予定

動作デモ

おわりに

- まとめ
  - Pro5Langをより多くの研究者の試行に供することを目的とした、Pythonによる新たな実装を行った
  - 構文に関する検討を行った
  - https://github.com/HidemotoNakada/pro5lang_publish.git で公開予定
- 今後の予定
  - 言語機能の整理
  - 構文の確定
  - UIの充実

謝辞

- 本研究はJSPS科研費JP18K11488, JP18K18117, JP22K12188の助成を受けたものです。