ソフトウェア基礎論　22－4
（１９８７．１０．１）

カットオペレータを排除し明示的に制御を行う論理型言語：ＡＬＥＸ

星田　昌紀　　所　真理雄

慶応義塾大学　理工学部　電気工学科

本論文では，Prologの宣言的なプログラム記述性の利点を生かしつつ，手続き的側面を強化したプログラミング言語「ALEX」を提案する。

ALEXでは，バックトラックを条件分岐に相当する"Retry"と，本質的な試行錯誤に相当する"Recall"という二つの範ちゅうに分け，この二つを別々の方法でコントロールする。そのコントロールは以下のように行う。

"Recall"はデフォルトで不可とし，「許可」を明示したときのみ可能とする。

"Retry" はデフォルトで可能とし，「禁止」を明示したときのみ不可とする。

このようにコントロールを明示的に記述することによって，カットオペレータを排除し，デバッグの容易化およびリーダビリティーの向上を図っている。また，バックトラックに制限を設けることにより「枝の刈り残し」によるバグも大幅に削減できる

しかもこれらの実行制御は，「述語」のプレフィクスとして記述されるため，プレフィクスを無視することにより　Prologと同様，プログラムの宣言的な解釈が可能である。

ALEX : THE LOGIC PROGRAMMING LANGUAGE WITH EXPLICIT CONTROL AND WITHOUT CUT-OPERATORS

Masaki HOSHIDA and Mario TOKORO

Dept. of Electrical Engineering Faculty of Science and Technology KEIO University

3-14-1 Hiyoshi, Kouhoku-ku, Yokohama, Kanagawa 223 JPN

We propose a Prolog-like new programming language "ALEX". It is an extention of Prolog from the view point of procedural programming , keeping the merit of declarative nature of Prolog . In ALEX, the backtrack mechanism is devided into two categories : "Retry" and "Recall". "Retry" corresponds to "branch on condition", while "Recall" coressponds to "essential try and error". And these two actions are controled in different methods. They are controled as follows.

"Recall" cannot be performed in default state ,and it can be performed only when permitted explicitly.

"Retry" can be performed in default state ,and it cannot be performed only when prohibited explicitly.

By such explicit expresion of execution control, "cut-operators" are excluded. As a result, we can achieve easy debugging and higher readability. And the restriction of the backtrack can effectively reduce the bugs of what we call "Branches forgotten to cut".

Further more , since these control is written as prefix of predicate , we can read ALEX programs declaratively as in Prolog by ignoring the prefix.

1　はじめに

Prologは，その特徴の一つとして試行錯誤的な実行機構をもち，宣言的な知識の表現に向いているが，そのためにプログラム実行時の制御がユーザから見て分かりにくくなっている。具体的には，「強力すぎるバックトラック」や「カットオペレータ」に起因する，デバッグのしにくさやリーダビリティーの低さが原因として挙げられる。

Prologプログラムの宣言性は，他言語には見られない大きな特長であり，その実現のためにバックトラックは必要である。しかし，プログラムの全ての部分に渡って宣言性が必要なわけではない。にもかかわらず，常にバックトラックがデフォルトとして存在し，ユーザが，決定的に実行したい部分（一般的プログラムでは，かなりの部分を占める）に対し，いちいちカットオペレータで枝刈りを行わなければならないのは自然とはいえない。バグの重要な原因にもなっている。

また他人の書いたPrologプログラムを読む際，カットオペレータと，クローズの排他関係だけをたよりに読み進むためリーダビリティーは低い。かなり下のレベルまで読まないと，どの述語が条件部かということすらわからないこともある。

さらに，Prologでは，「プログラムエラー」と「判断処理の結果としての失敗」が同一の振舞いをする。これはエラーの極在化を困難にし，エラー個所を探すために多量のトレースを必要とする。

プログラミング言語にとって重要なことは，ユーザの意図を正確に表現できることであり，原理的，理論的にどんなに優れていても，プログラミングがやりにくいのでは困る。現在Prologでカットオペレータだけが唯一の制御オペレータであるが，Prologを実用的なプログラミング言語と考えた場合，カットオペレータはPrologにとって必然のものではない。また優れたデバッガを作る試みもなされてはいるが，Prologが今後も使いやすい実用的なプログラミング言語として存在していくためには，デバッガだけでは不十分であり，Prologのコントロール仕様そのものを見直す必要があると筆者等は考えている。

本論文ではPrologの実行制御方式を改良するために試作したプログラミング言語「ALEX」(Advanced proLog with Explicit eXpression)を提案する。

2　導入

2－1　RecallとRetry

ALEXではバックトラックをRecall及びRetryいう２つの異なるカテゴリーに分類する。以下にその定義を行う。

Recall：
- 一度成功裏に終了した述語呼び出しに対して，もう一度別のクローズを試みるように要請すること。
- 兄弟及び兄弟の子孫へのバックトラックである。
- 本質的な試行錯誤に相当する。
- ある述語Pの兄弟とは，Pが，あるクローズのボディ中に存在するとき，そのボディ中のP以外の述語をいう。特にPの左側に存在する述語をPの兄ということにする。（A：－B，C，P，D．においてPの兄はB，C）
- ある述語Pの子孫とは，Pに直接的あるいは間接的に呼び出される全ての述語のことをいう。

以下にいくつかの例を挙げるが，その前に記号上の約束をしておく。

－記号上の約束－

大文字アルファベットは，クローズのヘッドあるいはボディに存在する各述語を表す。同じアルファベットで表される述語は，同じファンクタ及び同じアリティを持つとする。同じアルファベットで表される述語のうち，異なるサフィックスを持つものは，ファンクタ，アリティは同じだが引数のパタンだけが異なるとする。

例えば

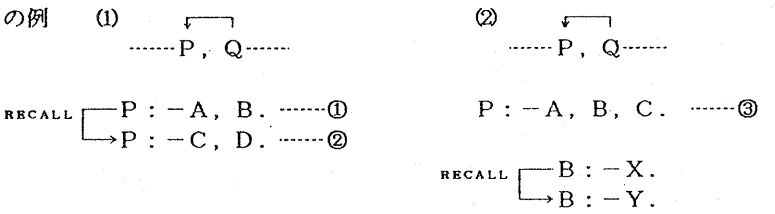
A_1．　　append([],L,L).

A_2：－A_3．　は　append([A|X],Y,[A|Z])：－append(X,Y,Z).

のようなパタンを表す。

またCaller述語とCallee述語は，おなじアルファベットで表されていても引き数のパタンが一般に異なるものとする。

<図1> Recallの例　(1)



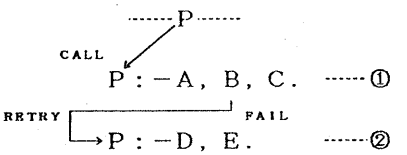
(1)は，述語Qの失敗により，Qの兄であるPにバックトラックがかかる場合である。Pの実行においては①が選択されていたとする。当然ではあるが，Qが実行されるということは，Pが成功裏に終了していることを意味する。このようなバックトラックをRecallと呼ぶことにする。A，Bが決定的に終了しているとすると，②が再試行される。このような場合を，「QからPにかかるRecall」と呼ぶ。

(2)も，(1)と同様，述語Qの失敗によりRecallが起こるが，Pそのものでなく Pから呼ばれているBに対してRecallがかかる。このときC，Xは決定的に終了しているものとする。このような場合を「Qから（Pの中の）Bにかかる Recall」と呼ぶ。この場合もBはすでに成功裏に終了している。

次にRetryの定義を行う。

Retry ：● ある述語の最初の呼び出しにおけるクローズの実行において，そのクローズ全体が失敗したために，他のクローズが選び直されること。
● 親へのバックトラックである。
● if then else,case等の条件分岐に相当する。
□ ある述語Pの親とは，Pを直接的に呼び出した述語をさす。

<図2> Retryの例



この例はPが呼び出されて①が選択され，次にAが決定的に終了し，その次のBが失敗した場合を想定している。このときBの親であるPの別の枝②が選び直されるが，この場合Pはまだ実行の途中であり成功裏に終了していない。このようなバックトラックをRetryという。

2－2　設計方針

先に，"Recall"は本質的な試行錯誤動作に相当し，"Retry"は条件分岐に相当することを述べた 。「条件分岐」（Retry)はプログラムにおいて非常に頻出するが，「試行錯誤動作」（Recall)はプログラムの全体量に比べてむしろ非常に少なくなるのが一般的傾向である。つまりRetryは普通起こって欲しいのだが，Recallの方は起こって欲しいことがまれである。一般に「まれにしか起こらない事態」に印を付けることが妥当だと考えられる。もしこの逆を行おうとすれば，「印の付け忘れ」が多発するであろう。

ところがPrologでは，この不自然なことを行っている 。 すなわちRecallのかかって欲しくない部分（一般的プログラムでは，かなりの部分を占める）にカットオペレータという「印」を付けなければならない。ゆえに「枝の刈り残し」によるバグが多発する。このようなことが起こる根本的な理由は，カットオペレータひとつでRecallとRetryの両方をコントロールしようとするところに起因している。

ALEXでは，RecallとRetryを別々の方法でコントロールする。そのコントロールは以下のように行う。

☞ Recallはデフォルトで不可とし，「許可」を明示したときのみ可能とする。
☞ Retry はデフォルトで可能とし，「禁止」を明示したときのみ不可とする。

Recallをデフォルトで不可とすることにより，「枝の刈り残し」によるバグを大幅に削減できる。

3 言語仕様

3－1 シンタクスとセマンティクスの概要

ALEXもPrologと同様，プログラムはクローズの集まりとして記述される。 クローズはPrologとほぼ同じ形をしているが，クローズのヘッド，及びボディ内の各述語は，プレフィクスを持つ可能性がある点が異なる。

ボディ内の各述語はコンマで区切られ，クローズの最後にはピリオドが置かれる。各述語はプレフィクスを持つ可能性があるという点を除いてPrologにおける述語と同じである。

ALEXではヘッドとボディは”＜－”で区切られる。

プログラムはPrologと同様，深さ優先で左から実行される。

3－2 Recallに関するコントロール(1)

ALEXにおける全てのクローズは，デフォルト状態において一度成功裏に実行が終了すると，そのクローズのオールタナティブは実行不可となる。但し，成功裏に終了したクローズの，ボディ中の述語に対してRecallをかけることは可能である。（これについては次節３－３〈図５〉の例で述べる。）

あるクローズＬが成功裏に終了したにもかかわらず，そのクローズＬのオールタナティブにRecallをかけたいときは，Ｌの先頭（ヘッドの前）に「：」を付記する。これによりＬ以下の（Ｌより後ろに書かれている）クローズが再試行の対象として生き残る。

ここで少しRetryについても述べておく。Retryはデフォルトで可能であるから，あるクローズ実行中に，そのクローズが全体として失敗したときは，次のクローズが（もし存在すれば）試される。Retryは「：」の有無に係わらず常に可能である。

以下に例をあげる。

〈図３〉

P_1＜－Ａ，Ｂ，Ｃ．‥‥‥‥‥①
：P_2＜－Ｄ，Ｅ．‥‥‥‥‥②
P_3＜－Ｆ，Ｇ．‥‥‥‥‥③

この例において仮に①が成功裏に終了したとすると，後にＰに対してRecallがかかることがあっても②③は試されない。つまり，②③が，あたかも存在しなかったかのようになる。これに対し，②が成功裏に終了すると，③はオルターナティブとして生き残る。つまり，後にＰに対してRecallがかかったとき，③が試されることになる。

3－3 Recallに関するコントロール(2)

ALEXではクローズのボディに存在する全ての述語を以下の３種類に大別する。（実際には５種類あるが残りの２種類は次節３－４で述べる。）

1）プレフィクスを何も持たないもの
- 必ず成功するものと見なされる。
- 失敗した場合，インタプリタはエラーメッセージを出して停止する。
- 実行終了後は，Recallを受け付けない。
- ロエラーメッセージは失敗した述語名と呼び出した変数のバインディング状況及びコールナンバーを表示する。
- ロエラーメッセージ表示の後インタプリタは，そのエラーを失敗と解釈して実行を続けるか，あるいは実行をアボートするかをユーザに問う。

2）「？」プレフィクスを持つもの
- 失敗する可能性を持つ 。
- 失敗した場合はRecallあるいはRetryを引き起こす。
- 実行終了後はRecallを受け付けない。

3）「：」プレフィクスを持つもの

- 失敗する可能性を持つ 。
- 失敗した場合はRecallあるいはRetryを引き起こす。
- 実行終了後，内部にRecallポイントがあればRecallを受け付ける。

□ある述語の内部とは，その述語によって直接的，間接的に呼び出された全ての述語群を指すとする。

□Recallポイントとは，Prologでいうところのバックトラックポイントのうち選択点以下が成功裏に終了しているもの（Retryポイントでないもの)に相当する。但しALEXの文法的制約（前節及び本節で述べられたもの）を満たしていることを前提とする。

□この「：」は，クローズのヘッドに付加される「：」(前節)とはセマンティクスが区別される。

以下に例を示す。

〈図4〉　　P<−:A，B，:C，?D，?E，F.

:A_1.　B.　:C_1.　D.　E.　F.
　A_2.　　C_2.

〈図4〉において，:Cが失敗したときは:Aにだけ，?Dあるいは?Eが失敗したときは:Cにだけ Recallがかかる。「:」プレフィクスの付加されていないBや?Dには，Recall がかからない。:CがRecallポイントを持たないときは，?Dあるいは?Eの失敗によって:AにRecallがかかる。さらに:AがRecallポイントを持たないときは，P<−……にオルターナティブがあるならば，そのオルターナティブが 試される。また，P<−……にオルターナティブがないならば，P自体 の失敗となる。

P自体が失敗したとき，もし，Pが ……P…… のように呼ばれてい れば,エラーメッセージを出して実行が停止する。また Pが ……?P…… あるいは ……:P…… のように呼ばれていれば,この例で?Eや:Aが失敗したときと同様のことが起こる。

〈図5〉　　……:P，Q，?R……

P_1<−A，:B，C. ……①
P_2<−D.　　……②

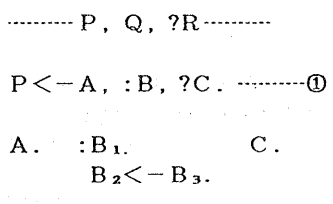
A.　:B_1.　　C.　D.
　　B_2<−B_3.

〈図5〉では,?Rが失敗したとき:PにRecallがかかるが，最初の:Pの呼び出しにおいて①が成功していたとすると，じっさいには:Pの内部の:BにRecallがかかる。

:BにRecallがかかったとき ，:Bが失敗したとしても②は実行されず:P自体が失敗する。なぜなら①はすでに成功裏に終了しているので，②はもはやオールタナティブではなくなっているからである。

この場合，もし，②をオールタナティブとして実行したいなら，①の先頭（Pの前）に「:」を付ける必要がある。このとき，:Pは内部に２つのRecallポイントを持つことになるが，最初にRecallがかかるのは:Bである。その理由は，:Bが:Pより最近のチョイスポイントだからである。これはPrologにおいても全く同じである。今述べた例は少し複雑なものであるが，このようなことは実際のプログラミングにおいてまれにしか起こらない。

〈図6〉



この例はALEXプログラムのモジュラリティーを示すものである。①において:B⇦?Cの間でRecallがかかって欲しいのだが，?C以外からは :BにRecallをかけたくないという状況を想定する。Prologのプログラミングにおいては，常に?Rのようなところから：Bにかかるバックトラックに注意してカットオペレータを挿入しなければならないのであるが，ALEXではその必要がない。つまりPはプレフィクスを持たないので，？Rから Pの内部にある:BにRecallがかかることはない。このようにPの内部は外部から遮断されている。一方:Bを 誤ってBと記述してしまった場合を仮定すると,?Cの失敗によってP全体が失敗し，Pのエラーになる。このようにP内部のエラーは局所化が図られる。Pが失敗せずに誤った値をバインドするといったようなバグは局所化できないが，意図せぬ失敗がプログラムの遠く離れた場所に影響を与えるということはない。

カットオペレータは，なぜそこに入れたのかという意味が分かりにくいことが多く，不必要なカットを挿入したプログラムも多く見受けられる。ALEXでは，このようなことはなくなりリーダビリティーも高くなっている。

ALEXでは，さらにRecallの局所化を図るために，ブロックという構造が用意されている。これについては３－５で述べる。

３－４Retryに関するコントロール

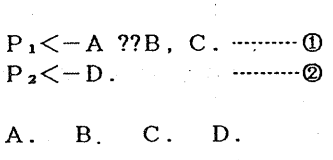
前述したようにRetryはデフォルト状態で可であり，Recallと同様に「？」あるいは「：」の付加された述語の失敗によって引き起こされる。これを禁止したいときには，ステージフェイルプレフィクスというプレフィクスを用いる。ステージフェイルプレフィクスは「?? 」および「::」で表される。

これらの付加された述語の性質を３－２節に追加するかたちで以下に定義する。

４）「??」プレフィクスを持つもの
- 失敗する可能性を持つ。
- 失敗した場合，親述語の呼び出しが即座に失敗する。
- 実行終了後は，Recallを受け付けない。

５）「::」プレフィクスを持つもの
- 失敗する可能性を持つ。
- 失敗した場合，親述語の呼び出しが即座に失敗する。
- 実行終了後は，「:」の付加された述語と同様，Recallを受け付ける。

〈図7〉



〈図7〉で ??Bが失敗すると ②を試さずに即座に?Pの失敗となる。このようにしてRetryを禁止する。この例から分かるように，あるステージ（すぐ後の〈補〉参照）においてRetryを起こさないようにした場合，同時にRecallも起こらなくなる。ALEXではRetryだけを禁止するオペレータは用意されていない。しかしPrologにおいてもRetryだけを禁止することは不可能であり，プログラミングにおいては,なんらさしつかえない。

〈補〉ある述語呼び出しPによって直接呼び出されるクローズ群（Pをヘッドに持つもの）のそのレベルでの実行をPのステージと呼ぶことにすると，「??」「::」の付加された述語は，それが存在するステージ全体を失敗させるので，これらのプレフィクスをステージフェイルプレフィクスと呼ぶことにする。
〈補〉「?」と「??」はそれぞれ単独で，フェイル及びステージフェイルを無条件に引き起こすオペレータとして利用できる。

以上ボディ中の５種類の述語を導入したが，プレフィクスの付加された述語を失敗可能述語，プレフィクスの付加されていない述語を失敗不可述語と呼ぶことにする。

３－５　ブロック

Recallのかかる範囲をさらに限定するためにブロックという構造を導入する。これは意図されていない場所からかかるRecallを防ぐための防御機構である。ブロックは「＜」と「＞」で囲まれた１つ以上の述語の並びである。
以下の性質を持つ。

- ブロックの外で起きた失敗によって引き起こされるRecallは，ブロックの中に入らない。ブロックをスキップして，一番最近のチョイスポイントに戻どる。
- ブロックの内で起きた失敗は外に出ない。つまり，ブロックが全体として失敗したときは，ブロックのエラーになる。

〈図８〉

```
P＜－A，＜:B，C，?D＞,E，?F．
P＜－G，:H．
```

:Bと?Dの間ではジェネレーション アンド テスト（Recall)が可能である。しかし，いったん?Dが成功すると，もはや:BにRecallがかかることはなくなる。例えば，?Fが失敗したとすると，次にGが実行される。:Bの兄弟だけでなく，上のレベルからのRecallからも:Bは守られる。

〈図９〉

```
----------:P，Q，?R----------
```

例えば，１つ上のレベルが上記のようになっていて，?Rが失敗したときも:BにRecallがかかることはない。（:HにRecallがかかる可能性はある。）
さらにブロックは以下の特徴を持つ。

- ブロックに全体としての失敗可能性を持たせたいときは「＜」の直前に「?」あるいは「?? 」を付加する。（意味は述語の場合と同じ）
- ブロックは入れ子構造をとることができる。

4　プログラム例

以下にPrologプログラムとALEXプログラムの対比例を示す。
〈図１０〉

```
-----Prolog-----
?-search(s(0,0),s(2,_),ANS).

search(X,_,_):-
      mark(X),!.fail.
search(X,X,[X]):-!.
search(X,GOAL,[X|ANS]):-
      asserta(mark(X)),
      ope(X,Y),
```

```
-----ALEX-----
?- :search(s(0,0),s(2,_),ANS).

search(X,_,_)<-
      ?mark(X),??.
search(X,X,[X]).
:search(X,GOAL,[X|ANS])<-
      asserta(:mark(X)),
      :ope(X,Y),
```

```
        check(Y),
        search(Y,GOAL,ANS).
search(_,_,_):-
        retract(mark(X)),!,fail.
ope(s(X,Y),s(X1,Y1)):-
        op(X,Y,X1,Y1).

op(X,Y,4,Y):-X<4.
op(X,Y,X,3):-Y<3.
op(X,Y,0,Y):-0<X.
op(X,Y,X,0):-0<Y.
op(X,Y,Z,0):-0<Y,Z is X+Y,Z=<4.
op(X,Y,0,Z):-0<X,Z is X+Y,Z=<3.
op(X,Y,4,R):-0<Y,4=<X,Y,R is Y-(4-X).
op(X,Y,R,3):-0<X,3=<X+Y,R is X-(3-Y).

check(Y):-pushTop,mark(X),ope(X,Y),
        popTop,!,fail.
check(Y):-popTop.

pushTop:-retract(mark(TOP)),!,
        assert(memo(TOP)).
popTop :-retract(memo(TOP)),!,
        asserta(mark(TOP)).
```

```
        ?check(Y),
        :search(Y,GOAL,ANS).
search(_,_,_)<-
        retract(:mark(X)),??.
ope(s(X,Y),s(X1,Y1))<-
                :op(X,Y,X1,Y1).

:op(X,Y,4,Y)<-?(X<4).
:op(X,Y,X,3)<-?(Y<3).
:op(X,Y,0,Y)<-?(0<X)
:op(X,Y,X,0)<-?(0<Y).
:op(X,Y,Z,0)<-?(0<Y),Z is X+Y,?(Z=<4).
:op(X,Y,0,Z)<-?(0<X),Z is X+Y,?(Z=<3).
:op(X,Y,4,R)<-?(0<Y),?(4=<X+Y),R is Y-(4-X).
 op(X,Y,R,3)<-?(0<X),?(3=<X+Y),R is X-(3-Y).

check(Y)<-pushTop,:mark(X),?ope(X,Y),
        popTop,??.
check(Y)<-popTop.

pushTop<-retract(:mark(TOP)),
        assert(memo(TOP)).
popTop <-retract(memo(TOP)),
        asserta(:mark(TOP)).
```

この２つのプログラムはＡＩの例題として有名な Water Jug Problem を解くためのものであり，両方とも同じ働きをする。それぞれ４リットル，３リットルずつ水の入る目盛りのない容器を，空の状態から初めて４リットルの方の容器に２リットルの水を計り取るという設定である。上記のプログラムは２つある最短解を両方求めることができる。ALEXでもPrologと同様，「；」を用いて別解を求める。探索プログラムであるため一般のプログラムに比べてかなりRecallの起こる部分は多くなっている。

ALEXではどのように再試行が起こるのかプログラムからすぐ読み取ることができリーダビリティーが高い。またデバッグ時において，例えばretractが失敗したとするとエラーメッセージがでるので：markがassertされていないとすぐ分かり，デバッグが効率的にできる。

check述語の内部には:markというRecallポイントがあるが，この述語は?checkとして呼ばれるため外から:markに対してRecallがかかることはない。この様にして，かなりのバグが削減される。

5 おわりに

冒頭でも述べたが，プログラミング言語にとって最も重要なことは，記述，デバッグ，保守がしやすいことである。Prologは大きな潜在的能力をもっているが，最も重要なプログラミングのしやすさを若干軽視していたきらいがある。またPrologの拡張言語も沢山作られているが，いくら機能が上がっても，デバッグにかかる時間が何倍にもなるものは実用的とはいえない。

ALEXは，ある種のバックトラックに制限を加えることと，制御を明示化することにより，分かりにくいといわれるPrologの実行制御を改善するひとつの方向を示した。まだできたばかりで十分なものとはいえないが，今後は実用的で大規模なプログラム等で，その有効性と問題点を検討していきたい。

参考文献

［１］沼尾　雅之　「Prologの視覚的デバッガ」情報処理学会　記号処理研究会資料No.32（1985）

［２］Lee Naish "PROLOG CONTROL RULES" IJCAI86 (1986)

［３］沢村　一　「Prolog述語（呼び出し）の決定性」ICOT Techinical Report TR-205 (1986)

［４］Clocksin,Merish "Programming in Prolog" Springer-Verlag (1981)