自然言語の語彙分割による形式的仕様記述†

佐伯元司†† 米崎直樹†† 榎本肇††

従来，プログラムの仕様は自然言語で書かれていたが，仕様を入力として，各種の文書合成や無矛盾性などの検証，さらにはプログラムの自動合成といった意味に関した処理を計算機で行うには，形式的意味をもつ言語により記述する以外に方法はない．一方，自然言語は人間にとってわかりやすいが，非形式的すぎる．本論文では，曖昧性のない擬似的な自然言語（英語）でプログラムの形式的な仕様記述を段階的に行う方法を提案する．われわれの仕様記述言語の意味的基礎は一階の述語論理である．仕様記述は，使用される単語の意味を，下位のレベルで定義された単語を用いた英文や論理式，さらには抽象データ型やそのオペレーションを用いて，定義しながら階層的に行っていく．このため，最上位レベルの仕様は人にとってわかりやすい英文で記述することができる．英文で書かれた仕様は，単語の意味定義から導かれる述語論理の有意式への変換規則に従って，論理式へと変換される．これにより，仕様記述内容に厳密な意味が割り当てられ，この論理の公理系のもとで，計算機による本格的な意味処理が可能になる．

1. まえがき

最近，生産されるソフトウェアの規模が大きくなるにつれて，そのソフトウェアの開発や保守に多くの人間が関わりあってくるようになってきた．その際，重要なことは，そのソフトウェアの仕様定義を行う人間が抱いているイメージを，他の開発者や保守を行う者にいかに正確に伝えるか，ということである．仕様記述言語はプログラミング言語などと違って人間の思考を正確に伝達するための道具としての役割が最も重要である．この役割を考慮すると，仕様記述言語としての望ましい条件として以下の項目が挙げられる．

1) 人間にとって理解が容易であること．一見しただけで内容の概略がわかること（伝達性）．

2) 曖昧性がないこと．すなわち，一つの記述から複数の意味が生じてこないこと（一義性）．

さらに，最近のようにソフトウェアの規模が大きくなると，仕様記述量も増大してくるため，

3) 機械的に仕様の意味を扱う処理（記述内容に矛盾がないか等の検証や文書合成）が可能であること．

また，人間が大規模システムを理解するには，階層的抽象化，モジュール化以外に方法がないから

4) 階層的な記述が可能であること，かつそのようなモジュール分解のガイドを与えうること（モジュラリティ）．

従来，仕様は自然言語で書かれていた．自然言語はわかりやすいというだけでなく，簡潔に表現できるという長所があるが，その反面，記述内容が曖昧になるという欠点がある．さらには，仕様の意味を扱うような処理を計算機で行うのには不適である．

これまで，計算機で処理できるほど形式的な仕様記述言語として，ISDOS/PSL[1]やSREM/RSL, R-net[2], SPECIAL[3], SADT 図式[4]などの言語が開発されてきたが，これらは

1) 各モジュールの仕様記述が，そのモジュールの制御，あるいはデータの流れを基礎としており，ひとまとまりの直観的な概念としてとらえにくい．また，必ずしも，われわれが一つの概念の単位として自然に考えうるモジュールに分解されていないため，一見しただけでは記述内容の概略がわかりにくい．

2) SPECIAL のように数学的な基盤の上で検証などの意味処理を行えるものもあるが，大部分は本格的な意味処理を行うには，数学的な基盤が不十分である．という欠点がある．ほかにも，本格的な検証を目的とし，形式論理や代数を基礎にした言語として，iota[5]，抽象データ型の代数的仕様記述言語[6]，HISP[7]などがあるが，記号論理式を直接読むことは，通常の人間にとって，慣れていないため困難である．

本論文では，上記のような問題についての新しい手法を提案する．われわれの仕様記述言語は曖昧性のない擬似的な自然言語（英語）である．仕様記述は，自然言語の文の集合として与えられ，そのなかで使用する単語の意味を階層的に自然言語を用いて定義していく．最も下位のレベルでは，単語の意味は抽象データ型のオペレーションの定義のように，単語間の関係記述として定義する．抽象データ型のオペレーションの

† Formal Specification Method Based on Lexical Decomposition of Natural Language by MOTOSHI SAEKI, NAOKI YONEZAKI and HAJIME ENOMOTO (Department of Computer Science, Faculty of Engineering, Tokyo Institute of Technology).

†† 東京工業大学工学部情報工学科

定義は自然言語や論理式を使って行われる．英文で書かれた仕様は，単語の意味定義から導かれる意味規則および各構文規則ごとに割り当てられた意味規則に従って，論理式へと変換される．これにより，仕様記述内容に厳密な意味が割り当てられ，さらにはこの論理の公理系のもとで，計算機による本格的な意味処理が可能になる．

2章では英文や論理式・抽象データ型を使って，仕様がどのように階層的に記述されていくかを，3章ではこの仕様記述言語の意味的基礎となる論理体系を，4章では単語の意味定義からどのようにして論理の有意式が生成されるかを，5章ではそれらの単語を使った英文で記述されたシステム全体の仕様が，どのようにして論理式へ変換されるかを述べる．

2. 仕様化技法

自然言語を用いて仕様を記述するために，以下のような三つの性質を自然言語がもっていると仮定する．

1) 仕様記述の対象となっているシステムを自然言語で説明することができる．そのとき用いた単語には，まだ明確にはされていないかもしれないが，まとまりをもった概念が対応している．

2) 単語のもっている概念には，ソフトウェアとしての，まとまりをもったモジュールが対応する．

3) 仕様記述に用いる単語として，その仕様を読むすべての人間に共通のイメージを与えるような単語を選択できる．

われわれは，まずシステムを自然言語で説明し，そのとき用いた単語に対応している漠然とした概念を引き出し，明確にしていく過程が仕様記述であると考える．言い換えれば，仕様記述とはそこで使用されている単語の意味を次々に詳しく説明していく過程である．この単語の意味の説明は，語彙分割 (lexical decomposition) によって，別の単語を使った自然言語で記述される．そして，その別の単語もまた，下位のレベルで同様にして定義されていく．最下位レベルの，すなわち基底となる単語は抽象データ型のオペレーションの定義方法のように，単語間の関係記述として定義される．各単語にはソフトウェアとモジュールが対応しているため，このように階層的に単語の定義を行っていくことは，ソフトウェアを階層的にモジュール分割していくことに対応している．仕様記述に使える文型を制限することにより，使用する単語のカテゴリを限定する．このことは，モジュール分割のさまざまなバリエーションを減らし，結果としてモジュール化の指針を与えてくれることになる．

動作のタイミング自身が問題となるようなプログラムを除くと，プログラムの本質的な仕様は，そのプログラムの入力・出力の関係を記述したものである．この関係は論理式の集合として表される．われわれの仕様記述方法では，擬似的な自然言語で入力・出力の関係を記述し，それを論理式に変換する．論理式への変換を行うために，前もって与えられた構文ごとの意味規則と単語ごとの意味規則がある．is や a, every などのような共通な単語では，その意味規則は前もって定義されている．そのほかの単語はユーザが定義し，その定義に従って意味規則が自動的に生成される．ユーザが定義できる単語は普通名詞と形容詞である．ここで対象としているプログラムは入出力関係だけで記述できるようなプログラムに限定しているため，普通名詞・形容詞と be 動詞で十分である．ユーザは，下位のレベルで定義される単語を使った英文や論理式で単語の意味を定義する．さらに，抽象データ型として宣言された型名やそのオペレーションを単語と対応づけることができ，それらの単語の意味はその抽象データ型の公理を使って定義される．抽象データ型モジュールは，一つのモジュールで関連のある複数の普通名詞・形容詞を定義するが，他は常に一つのモジュールが一つの普通名詞または形容詞に対応する．

上位レベルの単語の意味は，下位レベルの単語を自然言語として用いて記述されるが，このような定義の階層性により，記述の上位レベルでは，下位レベルで定義すべき単語の厳密な意味を隠して，その単語を自由に自然言語として使用できる．その結果，読者は，仕様を一見しただけで，その記述内容の概略を理解することができる．それは，

1) ソフトウェアは，一般に現実の世界をモデル化したものと見なすことができ，自然言語の単語には現実世界のオブジェクトのイメージが付加されていると考えられる．したがって，ソフトウェア・モジュールに単語を対応させることにより，単語がうまく選択されていれば，ソフトウェア・モジュールとしてモデル化されたもののイメージを即座に思い浮かべることができる．

2) 仕様記述中に，まだ未知の単語が含まれていても，その単語のもつイメージと文中の既知の単語の出現より，その文の意味するもののイメージを文章理解として即座にもつことができる．

3) 未知の単語のイメージが漠然としていても，それを用いた仕様記述の意味するイメージをあらかじめ思い浮かべることができ，そのイメージをもとにして未知の単語の意味定義を読んでいくことにより，理解が容易になる．
という理由による．

本仕様化技法で用いた自然言語は，文法を単純なものに限定しているため，ごく自然な意味については，予備知識のない読者が通常の文として読んでも誤って理解してしまうことはほとんどない．また，5章で述べるように論理式への翻訳過程において，二つ以上の意味が生じないように意味規則を構成したため，機械処理に関しても曖昧性はなく，このような意味規則はわれわれの自然言語理解に対応するものである．

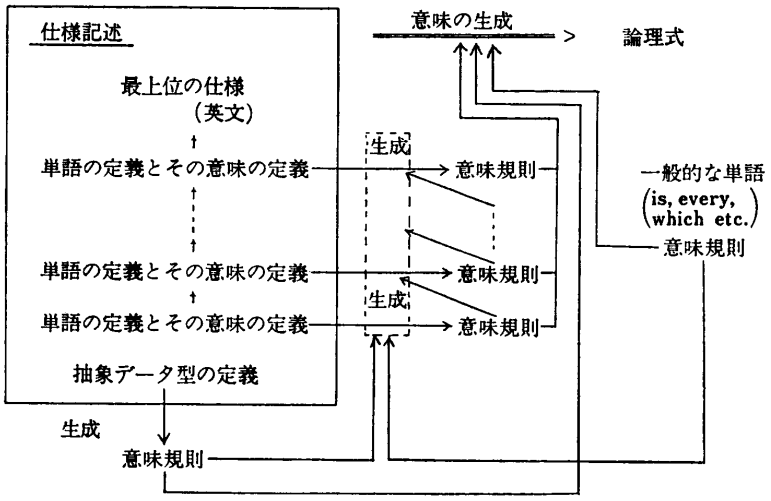


図 1 仕様化技法
Fig. 1 Specification technique.

図1に本仕様化技法のブロック図を示す．

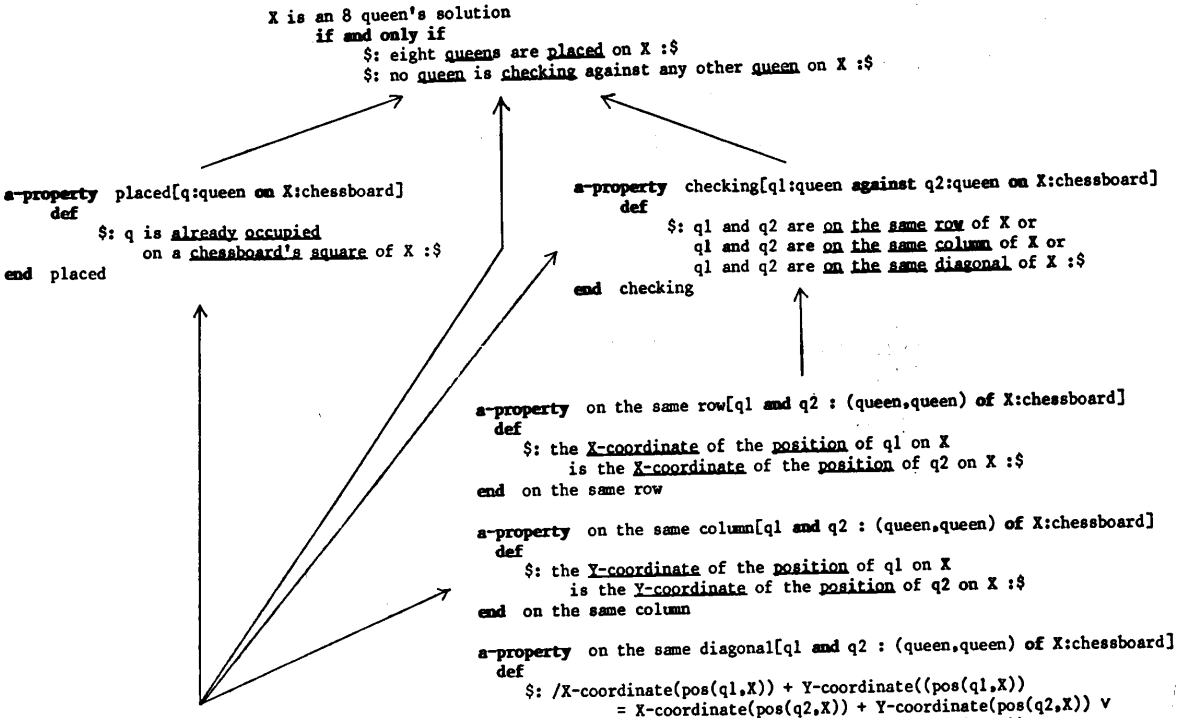
以下で実際のプログラムを例にとり，仕様化技法を説明する．

(8クイーン問題) 8クイーン問題，すなわち，8個のクイーンを互いに他を獲ることができないように，チェス盤の上に配置する問題を解くプログラムの仕様記述例を**図2**(a)に示す．単語の定義は，a-property…end ブロックと抽象データ型宣言部 type の op ブロックで行われている．最上位の仕様は，以下のようなことを述べている．「8 queen's solution は，(1)八つの queen が置かれていること．(2)どんな queen も互いに他を捕獲し合わないこと．といった条件を満足する queen の配置である．」これは，8クイーン問題の最も本質的で自然な定義である．この英文中に現われる特有の単語，たとえば，queen, placed, checking といった単語の厳密な意味は，下位のモジュールで定義される．たとえば checking の定義は，英文を使って，その語の意味を詳しく説明する形で行われている．この定義によると，二つの queen が chessboard 上の同じ行にあるか，同じ列にあるか，同じ対角線上にあるとき，片方の queen がもう一方を捕獲しているという．この checking の意味定義で使用された on the same row, on the same column, on the same diagonal といった単語の意味も同様に下位のレベルで定義されていく．

予約語 a-property は形容詞を定義するために使用される．普通名詞を定義するときは c-property を使用する．各 property ブロック中の $: … : $ で囲まれている部分は，一つの論理式あるいは英文である．この節がブロック中に複数個あるときは，互いに conjunctive に結合される．この記法は箇条書き形式であり，このような形式を取り入れたのは，たとえ英文でも長く続けて書くと，記述内容を理解することがむずかしくなるためである．property ブロックの仮引数リスト中の前置詞(たとえば，a-property checking… 中の前置詞 against など) は引数のセレクタとして働き，この単語が英文中に用いられたときに，修飾句として使用する前置詞を表す．前置詞は，普通名詞や形容詞とは違って，単語としての意味はもっていない．

抽象データ型としては，チェス盤 (chessboard)，チェス盤のます目 (chessboard's square)，クイーン (queen) が定義され，これらのモジュール内で，queen を置く動作 (occupy)，置かれた queen の位置 (pos)，queen が置かれているかどうか (isoccupy)，チェス盤のます目の座標 (X-coordinate, Y-coordinate) といった概念が抽象データ型のオペレーションとして定義され，これらのオペレーションと単語とを対応づけることが行われている．たとえば，isoccupy というオペレーションは，already occupied という形容詞と対応づけられる．

図2(b)は，最上位レベルの仕様，すなわち二つの英文を4章，5章で述べるような手法で論理式に変換



(a) 仕様　　(b) 論理式

図 2 8クイーン問題
Fig. 2 Eight queen problem.

```
type  set[of α : type]  ({α})
    operation
        comp : {α} → {α}
          comp[A]  (~A) : :complement of A [CN]
        union : ({α},{α}) → {α}
          union[A,B]  (A ∪ B): :union of A and B [CN]
        intersection :  ({α},{α}) → {α}
          intersection[A,B]  (A ∩ B): :intersection of A and B [CN]
        empty :  → {α}
          empty[]  ( φ ): :empty-set [CN]
        make-set : α → {α}
          make-set[x]  ( {x} ): :made from x [Adj]
        element : (α,{α}) → bool
          element[p,A]   ( p∈A ): :element of A [CN] , included in A [Adj]
        subset : ({α},{α}) → bool
          subset[A,B]  (A ⊆ B): :subset of B [CN] , included in B [Adj]
        number : {α} → integer
          number[A]: :element-number of A [CN]
        $: / element[p,A] ↔ ~element[p,comp[A]] / :$
        $: / element[p,union[A,B]] ↔ element[p,A]∨element[p,B] / :$
        $: / element[p,intersection[A,B]] ↔ element[p,A]∧element[p,B] / :$
        $: / ~element[p,empty[]] / :$
        $: / element[p,make-set[q]] ↔ p=q / :$
        $: / subset[A,B] ↔ ∀p[element[p,A] → element[p,B]] / :$
        $: / number[empty[]]=0 / :$
        $: / ~element[p,A] ↔ number[union[A,make-set[p]]=number[A]+1 / :$
        $: / element[p,A] ↔ number[union[A,make-set[p]]=number[A] / :$
    endop
end  set
```

図 3 集合データ型の公理
Fig. 3 Axioms for sets.

したもので，これが仕様の厳密な意味である．

number は集合データ型の要素数を数える関数で図3の公理によって定義される．

3. 論理体系

われわれの仕様記述言語の意味的基礎は一階の述語論理であり，英文で書かれた仕様はすべて論理式へ変換される．ここでは，この論理体系について，ごく簡単に述べる．

[タイプ] タイプの集合 T は以下の条件を満たす最小集合である．

1) 任意の文字列（ただし '〈' や '〉' は含まない）$\in T$（基本タイプ）

たとえば，t, integer, char, stack などはタイプである．

2) $\alpha_1, \alpha_2 \in T$ ならば $\langle \alpha_1\ \alpha_2 \rangle \in T$（関数タイプ）

[有意式] 有意式は通常の λ 記号をもつ一階述語と同様に，変数や定数，true, false, ～（否定記号），=（等号），∧（論理積），∨（論理和），→（含意），∀（全称記号），∃（存在記号），λ（抽象化記号），"(" ")"（関数適用）を通常の意味で使用する．ただし，∃ や ∀ は基本タイプの変数しか束縛できないが，λ は任意の変数のタイプを束縛することができる．便宜上の記法として，$A(B_1)\cdots(B_n)$ を $A(B_1, \cdots, B_n)$ と書く．また +，−，<，∈などの関数の適用に対しては，$+(x)(y)$ と書かずに，通常のように $x+y$ と infix notation で書くことにする．

4. 単語の意味定義とその有意式への翻訳

仕様記述に用いる単語の意味定義は抽象データ型・論理式・英文を使って行う．定義された単語は，この節で述べる方法により，論理の一つの有意式に翻訳され，その単語が英文に使用されたときは，5章で述べるようにその有意式を使って，英文が一つの論理式に変換される．ここで抽象データ型を用いるのは，それが抽象化の単位として自然でわかりやすく，そのデータ型名やオペレーションが，自然言語のあるカテゴリの単語と自然に対応がつくからである．

4.1 抽象データ型

本仕様記述言語中で使われる抽象データ型は，4.1.1 項で述べるような構造式を用いて記述する方法と Guttag らが提案しているように[6]，オペレーションの公理の集合として記述する方法の2種類がある．構造式を用いる理由は，オペレーションの公理を自然言語で記述することもできるが，その自然言語よりも式のほうがわかりやすいからである．構造式で抽象データ型が宣言された場合，そのデータ型の意味・すなわちオペレーションの公理は，構造式の構文に応じた公理のスキーマがあり，それを用いて自動生成される．

4.1.1 構造式

構造式の集合 Tf は以下の条件を満たす最小集合として定義される．

1) （基本構造） すでに抽象データ型として定義された型名および INTEGER, BOOL, CHAR は Tf の元である．

2) （関数型） $\alpha_1, \alpha_2 \in Tf$ ならば $\alpha_1 \to \alpha_2 \in Tf$

3) (集合型) $\alpha \in Tf$ ならば $\{\alpha\} \in Tf$

4) (直積型) $\alpha_1, \cdots, \alpha_n \in Tf$, $a_1, \cdots, a_n$ を任意の文字列 (ただし, $i \neq j$ においては $a_i \neq a_j$ とする) ならば,

$(a_1 : \alpha_1, \cdots, a_n : \alpha_n) \in Tf$

ここで, a_i $(1 \leq i \leq n)$ は省略可能である.

5) (直和型) $\alpha_1, \cdots, \alpha_n \in Tf$, $a_1, \cdots, a_n$ を任意の文字列 (ただし $i \neq j$ においては $a_i \neq a_j$ とする) ならば

$(a_1 : \alpha_1 | \cdots | a_n : \alpha_n) \in Tf$

ここで, a_i $(1 \leq i \leq n)$ は省略可能である.

6) (列挙型) v_i $(1 \leq i \leq n)$ をデータ型 α の値とすると

$[v_1, \cdots, v_n] \in Tf$

また連続する整数値などのように, その下限値 v_1, 上限値 v_n だけで途中の値 $v_2, \cdots, v_{n-1}$ がわかるものについては, $[v_1 .. v_n]$ と書く.

2)～6) はすでに定義されているデータ型やそれに属する値を使って新しいデータ型を定義するためのものである.

4.1.2 データ型と論理のタイプとの対応

宣言された抽象データ型の意味は, そのオペレーションの公理, すなわち論理式の集合として記述される. ここでは, まず抽象データ型と論理のタイプとがどのように対応し, それを使って抽象データ型の意味はどのような論理式で記述されるかを述べる.

データ型を論理のタイプに対応づける関数 f を以下のように定義する.

1) $f(\text{INTEGER}) = \text{integer}$,
$f(\text{CHAR}) = \text{char}$, $f(\text{BOOL}) = t$

2) α を宣言された抽象データ型名とする.

2-1) α が構造式を用いずにオペレーションの公理だけで宣言されているとき, $f(\alpha) = \alpha'.\alpha'$ は α を英小文字で表したもので, これは基本タイプである.

2-2) α が構造式を用いて宣言されているときはその構造式により, $f(\alpha)$ は以下のようになる. この場合もすべて基本タイプに変換される.

$f((a_1 : \alpha_1, \cdots, a_n : \alpha_n)) = \text{tuple of } \alpha_1, \cdots, \alpha_n$
$f((a_1 : \alpha_1 | \cdots | a_n : \alpha_n)) = \text{sum of } \alpha_1, \cdots, \alpha_n$
$f(\{\alpha_1\}) = \text{set of } \alpha_1$
$f(\alpha_1 \to \alpha_2) = \text{func } \alpha_1\ \alpha_2$
$f([v_1, \cdots, v_n]) = f(\alpha_1)$

ただし, v_i $(1 \leq i \leq n)$ のデータ型は α_1 とする.

構造式は再帰を含んでいてもかまわない. ここで, 集合型 $\{\alpha_1\}$ を $\langle \alpha_1\ t \rangle$, 関数型を $\langle \alpha_1\ \alpha_2 \rangle$ ではなくて, 基本タイプに変換するのは仕様記述の論理式を一階にするためである. 直積型, 直和型も同じ理由で基本タイプに変換される.

また, オペレーションの型もこの関数 f により論理におけるタイプに変換される.

次に抽象データ型の宣言から, その意味としてどのような論理式が生成されるかを述べる.

1. 抽象データ型が構造式を用いずに, オペレーションの公理の集合だけで定義されているときは, 生成の必要はない.

2. 構造式が使用されているとき

以下に示すような構造式の構文に応じて, 公理のスキーマが用意されており, 公理のスキーマから生成された論理式を付け加えていく.

1) INTEGER, BOOL, CHAR に対しては, 通常のオペレーション (たとえば, INTEGER では, 0, +, −, +1, −1, ≦ など) とそれに関する公理があらかじめ用意されている. すでに, 抽象データ型として宣言された型名に対しては, それが宣言されたときの公理そのものである.

2) 直積型 $(a_1 : \alpha_1, \cdots, a_n : \alpha_n)$ に対しては, 新たに i 番目の要素を取り出す selector a_i とこの型のデータを作る constructor ( ) がオペレーションとして定義される. これらのオペレーションの意味を定義するために以下のような公理のスキーマが用意されている.

$$\forall x_1 \cdots \forall x_n [a_i'((x_1, \cdots, x_n)) = x_i]$$

ただし, x_i $(1 \leq i \leq n)$ はタイプ $f(\alpha_i)$ の変数で, a_i' は a_i と字づらが同じで, タイプ $\langle \text{tuple of } \alpha_1, \cdots, \alpha_n, f(\alpha_i) \rangle$ の非論理定数, constructor ( ) はタイプ $\langle \alpha_1 \langle \alpha_2 \langle \cdots \langle \alpha_n\ \text{tuple of } \alpha_1, \cdots, \alpha_n \rangle \cdots \rangle$ の非論理定数である.

3) 直和型 $(a_1 : \alpha_1 | , \cdots, | a_n : \alpha_n)$ (以下たんに β と書く) に対しては, 新たに inspector a_i $(1 \leq i \leq n)$ がオペレーションとして定義される. a_i は, β 型の値が α_i 型でもあれば true を返す述語である. このオペレーション以外に, データ型 α_i で定義されたオペレーションもタイプが変わるだけでそのまま有効になる. すなわち α_i 型のオペレーションで, constructor はその値域のタイプを $f(\beta)$ に, その他のオペレーションはそのタイプの $f(\alpha_i)$ の出現をすべて $f(\beta)$ で置き換えたタイプをもつオペレーションとして, 新たに

定義される．このオペレーションをα_i型からβ型に拡大されたオペレーションという．この公理はα_i型のそれと同じである．それ以外に生成される公理のスキーマは，α_i $(1\leq i\leq n)$ のすべての constructor$_i$ に対し

$$\forall x_1\cdots\forall x_n[a_i'(\overline{\text{constructor}_i}(x_1,\cdots,x_n)]$$

ただし，a_i'はタイプ$\langle f(\beta)\ t\rangle$の非論理定数，$\overline{\text{constructor}_i}$は，constructor$_i$を$\beta$型に拡大したもの，$x_1,\cdots,x_n$はオペレーションの引数としての自由変数である．

4) 集合型 $\{\alpha\}$ に対しては，図3で示すようなオペレーションの公理が生成される．

5) 関数型 $\alpha_1\rightarrow\alpha_2$ に対しては，constant, update という二つの constructor と apply が新たにオペレーションとして定義される．

$$\forall x\ [\text{apply (constant }(a),\ x)=a]$$
$$\forall x\ [x\neq b\rightarrow \text{apply (update }(f,b,a),\ x)$$
$$=\text{apply }(f,x)$$
$$\wedge x=b\rightarrow \text{apply (update }(f,b,a),\ x)=a]$$

ここで，x, bはタイプ $f(\alpha_1)$, aはタイプ $f(\alpha_2)$ の変数，constant はタイプ$\langle f(\alpha_2)$, func $\alpha_1\alpha_2\rangle$, update は$\langle$func $\alpha_1\alpha_2\langle f(\alpha_1)$, $\langle f(\alpha_2)$, func $\alpha_1\alpha_2\rangle\rangle\rangle$, apply は$\langle$func $\alpha_1\alpha_2\langle f(\alpha_1)$, $f(\alpha_2)\rangle\rangle$の非論理定数である．すなわち，constant は値域の値を常に a とする定数関数を作り，update は b の写像された先が a で，他はすべて f と一致する関数を作るオペレータである．apply は関数の適用を行うオペレータである．

6) 列挙型 $[v_1,\cdots,v_n]$ に対しては，v_i' $(1\leqq i\leqq n)$ 自身を constructor と考える．

$$\forall[x=v_1\wedge\cdots\wedge x=v_n]$$

図4 (b)は list of a の構造式から生成された公理の集合である．

4.2 単語の翻訳

自然言語の単語は，各 property によって定義されたり，抽象データ型の型名やオペレーションと対応づけて定義される．ここでは，そのような単語の意味規則，すなわち，論理の有意式への翻訳規則について述べる．

```
type  list[of a:type]
    def
        $: (empty: [nil] | compl: (car: a, cdr: list[of a])) :$
end  list
```

(a) 構造式を用いた定義

```
empty(nil)
∀a∀ycompl((a,y))
∀a∀ycar((a,y))=a
∀a∀ycdr((a,y))=y
```

(b) 生成される意味規則

図4 抽象データ型 list
Fig. 4 Abstract data type *list*.

4.2.1 property で定義された単語の翻訳

ユーザが英文や論理式を使って定義した単語は，以下のようにして，その意味規則が生成される．

1) $: …: $ 節内の英文をすべて論理式に変換する．変換方法は5章で述べる．

2) property ブロック中にある $: …: $ 節から得られた論理式をすべて∧で結合する．

3) 2)で生成された論理式の仮引数をすべてλ抽象する．

たとえば図2(a)の a-property checking…の場合は唯一の $: …: $ 節中の英文を論理式に翻訳すると，

$$\text{on the same row}'\ (q_1)(q_2)(X)$$
$$\vee\ \text{on the same column}'\ (q_1)(q_2)(X)$$
$$\vee\ \text{on the same diagonal}'\ (q_1)(q_2)(X)$$

になり，この論理式を仮引数 q_1, q_2, X でそのまま λ 抽象すると，

$$\lambda q_1\lambda q_2\lambda X\ [\text{on the same row}'\ (q_1)(q_2)(X)$$
$$\vee\ \text{on the same column}'\ (q_1)(q_2)(X)$$
$$\vee\ \text{on the same diagonal}'\ (q_1)(q_2)(X)]$$

となる．これが形容詞 checking の意味規則である．ただし，a' は単語 a の有意式への翻訳を示すものとする．

4.2.2 抽象データ型に対応づけられた単語の翻訳

単語の意味を抽象データ型の型名やオペレーションに対応させて定義したとき，その意味規則は以下のようにして自動的に生成される．

1) 抽象データ型名に対応させたとき

データ型名は常に普通名詞としか対応しない．とくにどの普通名詞と対応させるか記述されていないときは，型名自身が対応する単語となる．この単語の意味規則は，抽象データ型がαのとき

$$\lambda x_{f(\alpha)}[\text{true}]$$

となる．これは，この単語を含む語句の有意式のタイプを決定するのに使われる．

2) オペレーションに対応させたとき

オペレーションと対応できる単語は普通名詞，形容詞のどちらでもよく，その区別は，単語の後に [CN] もしくは [ADJ] を付けて行う．その単語の意味は，対応するオペレーションをA，そのタイプを$\langle\alpha_1\langle\cdots\langle\alpha_n\ \beta\rangle\cdots\rangle$

とすると

a) $\beta=t$ のとき

$\lambda x^1_{\alpha_1}, \cdots, \lambda x^n_{\alpha_n}[A(x^1, \cdots, x^n)]$

b) $\beta \neq t$ のとき

$\lambda x^1_{\alpha_1}, \cdots, \lambda x^n_{\alpha_n} \lambda y_\beta[A(x^1, \cdots, x^n)=y]$

である．オペレーションAの意味は，抽象データ型の定義から，前節で述べた方法で生成された論理式で記述される．たとえば，図2(a)の普通名詞 *X*-coordinate of の意味規則は，chessboard's square に対するオペレーション *X*-coordinate を使って，$\lambda x \lambda y$ [*X*-coordinate $(x)=y$] と生成される．この式中に出現しているオペレーション *X*-coordinate の意味は，直積型の意味規則 $\forall x_1 \forall x_2$ [*X*-coordinate$((x_1, x_2))=x_1$] で表される．

5. 英文の論理式への翻訳

仕様記述に使用された英文は，各単語の意味規則，すなわち，有意式への翻訳をもとに，自然言語の論理分析を目的とした理論であるモンテギュー文法[8]の考え方を応用した手法で論理式に変換される．変換規則は英文の構文ごとに用意されている．ここで使用できる英文は，be 動詞や関係代名詞を使った限定された平叙文である．ユーザが定義した以外の一般の単語，たとえば is, every, a, not などの意味規則は，あらかじめ用意されている．これらの単語は，すべて有意式への翻訳をもっているのではなく，関係代名詞 such that や which，前置詞のように単語として翻訳をもたず，syncategorematic に導入された単語もある．関係代名詞は，構文上でしか意味をもたないが，前置詞は，構文上の意味以外に，前置詞句で修飾される単語の仮引数と実引数の対応を制御する役割がある．

ここで用いた変換手法とモンテギュー文法のそれとの相違点は二つある．一つはこの前置詞の機能でありもう一つは，単語や句・節を有意式に翻訳した際，そのタイプは syntactic カテゴリに応じて，一意には決まらないことである．有意式のタイプは，その単語を修飾する句や節のもっているタイプや修飾の数に応じて変化する．単語が修飾を受けると，λ 変換が起こり，代入する有意式のタイプにより，λ 抽象された有意式の generic なタイプとして表されていたタイプが決定する．このような generic なタイプと修飾句に対応する λ 抽象の数を可変なものとして表現すると，タイプのスキーマは syntactic カテゴリに応じて一つのタイプスキーマで記述できる．これを**表1**に示す．

仕様記述に使用できる英文の構文規則とそれに付随する変換規則を付録に示す．変換規則，すなわち文・節・句・単語の意味規則は構文規則の右辺の項に対応する有意式から左辺の項に対応する有意式を求めるための規則で，これも論理の有意式として記述される．付録で示した構文規則・意味規則は次のような記法で記述される．

1) 各構文規則と意味規則は，以下のように1対1に対応している．

非終端記号＝導出規則1
　　意味規則1（有意式）
……
　　導出規則 n
　　意味規則 n（有意式）

2) 構文規則は

$A=B_1(E_1)B_2(E_2)\cdots B_n(E_n)$

の形とする．B_i は非終端記号もしくは終端記号，E_i は B_i を有意式へ翻訳したものとする．この構文規則に対応する意味規則には E_i を用いて A に対応する有意式を意味として生成する計算規則が書かれる．こ

表1 syntactic カテゴリとタイプとの関係
Table 1 Relations of syntactic categories and types.

Syntactic category	Type	Example
⟨sentence⟩	t	no queen is checking against any other queen on X
⟨term⟩$_\alpha$	⟨⟨α t⟩ t⟩	no queen, any other queen, X
⟨be-verb⟩$_\alpha$	⟨⟨⟨α t⟩ t⟩ ⟨α t⟩⟩	is , is not
⟨determinar⟩$_\alpha$	⟨⟨α t⟩ ⟨⟨α t⟩ t⟩⟩	a , every , no , the
⟨determinar 2⟩$_\alpha$	⟨⟨α t⟩ ⟨⟨α ⟨α t⟩ t⟩⟩⟩	any other
⟨noun clause⟩$_\alpha$	⟨α t⟩	queen, *X*-coordinate , *Y*-coordinate
⟨adjective phrase⟩$_{[\alpha_1, \cdots, \alpha_n]}$	⟨α_1 ⟨$\cdots$⟨α_n t⟩$\cdots$⟩	checking (n=3), placed (n=2)
⟨adjective phrase⟩$_{[\alpha]}$	⟨α t⟩	checking against any other queen on X
⟨connective⟩	⟨t ⟨t t⟩⟩	and , or

こで B_i に対応する E_i がなくてもよい．この場合は B_i は syncategorematicな項で意味規則では使用されない．

3) 非終端記号や終端記号の右下に付けられた添字 α (たとえば $\langle term\rangle_\alpha$) は，その翻訳となる有意式を決めるためのタイプを表す．その有意式のタイプは表1に示すような syntactic カテゴリ別のタイプのスキーマによって決定される．たとえば $\langle term\rangle_{integer}$ カテゴリに属する単語・句・節を翻訳した際の有意式のタイプは，すべて $\langle\langle integer\ t\rangle t\rangle$ となる．このgeneric なタイプ α は，構文解析が終わった時点で一意に決まる．関連するタイプが複数のときは，それらのタイプを [⋯] で囲んで記述する．たとえば $\langle adjective\ phrase\rangle_{[\alpha_1,\cdots,\alpha_n]}$ は，その形容詞句が n 個の引数をもち，それぞれのタイプが $\alpha_1,\cdots,\alpha_n$ であることを示している．このように複数のタイプが関係してくるのは，前置詞句により修飾を受ける普通名詞や形容詞を翻訳する場合と，関係代名詞節を翻訳する場合である．複数タイプの添字をもつ syntactic カテゴリは，他のカテゴリから修飾を受けたり，他のカテゴリを修飾したりするたびに，λ 抽象されている変数の数が一つずつ減っていく．構文規則の上では，添字 $[\alpha_1,\cdots,\alpha_n]$ 内の要素数が一つずつ減っていき，(付録の構文規則23, 27, 45参照)，やがて一つになる．前置詞句で修飾される場合はどの変数が λ 変換によって消去されるかが決まる．たとえば，checking against any other queen という形容詞句では，against という前置詞より checking の翻訳 $\lambda x\lambda y\lambda z$ [checking′ $(x)(y)(z)$] の第2引数 y に any other queen の翻訳 $\lambda b\exists u\exists v[u\neq v\wedge b(u)(v)]$ が対応することがわかる．付録の規則では，第2引数 y がヨで束縛され (規則45参照)，

$$\lambda x\lambda z\exists y\,[\text{checking}'(x)(y)(z)\wedge\lambda b\exists u\exists v\,[u\neq v\wedge b(u)(v)](\lambda y_1\lambda y_2[y_1=x\wedge y_2=y])$$

となり，これを簡略化すると

$$\lambda x\lambda z\exists y[x\neq y\wedge\text{checking}'(x)(y)(z)]$$

となる．これが形容詞句 checking against any other queen の翻訳である．

4) $\langle name\rangle_\alpha$ は，仕様記述中に出現する定数や変数で，そのデータ型はタイプ α と対応している．これらは，そのままタイプ α で字づらも等しい定数や変数に変換される．これを $name_\alpha'$ と書く．

英文の実際の翻訳過程を構文解析木の形で**図5**に示す．各枝に付けられた番号は，適用された構文規則および意味規則を表す．解析木のノードに付けられている有意式は，各構文規則に対応した意味規則によって生成されたものにこの論理体系の公理や定理を適用し，簡略化したものである．

6. むすび

本報告では，プログラムの仕様記述を擬似的な自然言語で行うことを提案し，そのために新しい仕様化技法を述べた．この手法により，理解性（わかりやすいこと）と形式性（数学的に意味が厳密に定義されること）がともに優れた仕様記述が行えると考えている．さらに，よく使用される単語の定義をデータベース化することにより，常識的な知識を隠蔽して仕様記述を行うことができる．

今後の課題としては，以下のようなことが挙げられる．

1) 擬似自然言語で記述できるとはいえ，検証などを行うために必要な記述量は，従来の形式論理をベースにしている仕様記述言語と変わりはない．指示語の使用など文脈情報を活用した自然言語特有の簡潔な記述ができることが必要であろう．このための論理の枠組として，文脈情報を扱えるような論理[9]や非単調な論理[10),11)]の導入が期待される．

2) 並列プログラムや通信プロトコルなど時間概念が重要な要素であるようなシステムの仕様記述においては，動詞を定義できる機能が必要である．そして，意味的基礎を述語論理ではなく，時間論理[12)]に拡張しなければならない．

3) 仕様記述内容の無矛盾性や各種の性質の検証は，本仕様記述言語のベースとなっている形式論理の公理系で行うことができるが，これを自動的に行うための強力な prover が必要である．

4) 英文で記述した仕様を論理式を仲介として，他の言語へ翻訳したり，論理式から，ある決められた format に従った英文の文書（ドキュメント）を合成したりする[13)]ことも興味ある課題である．

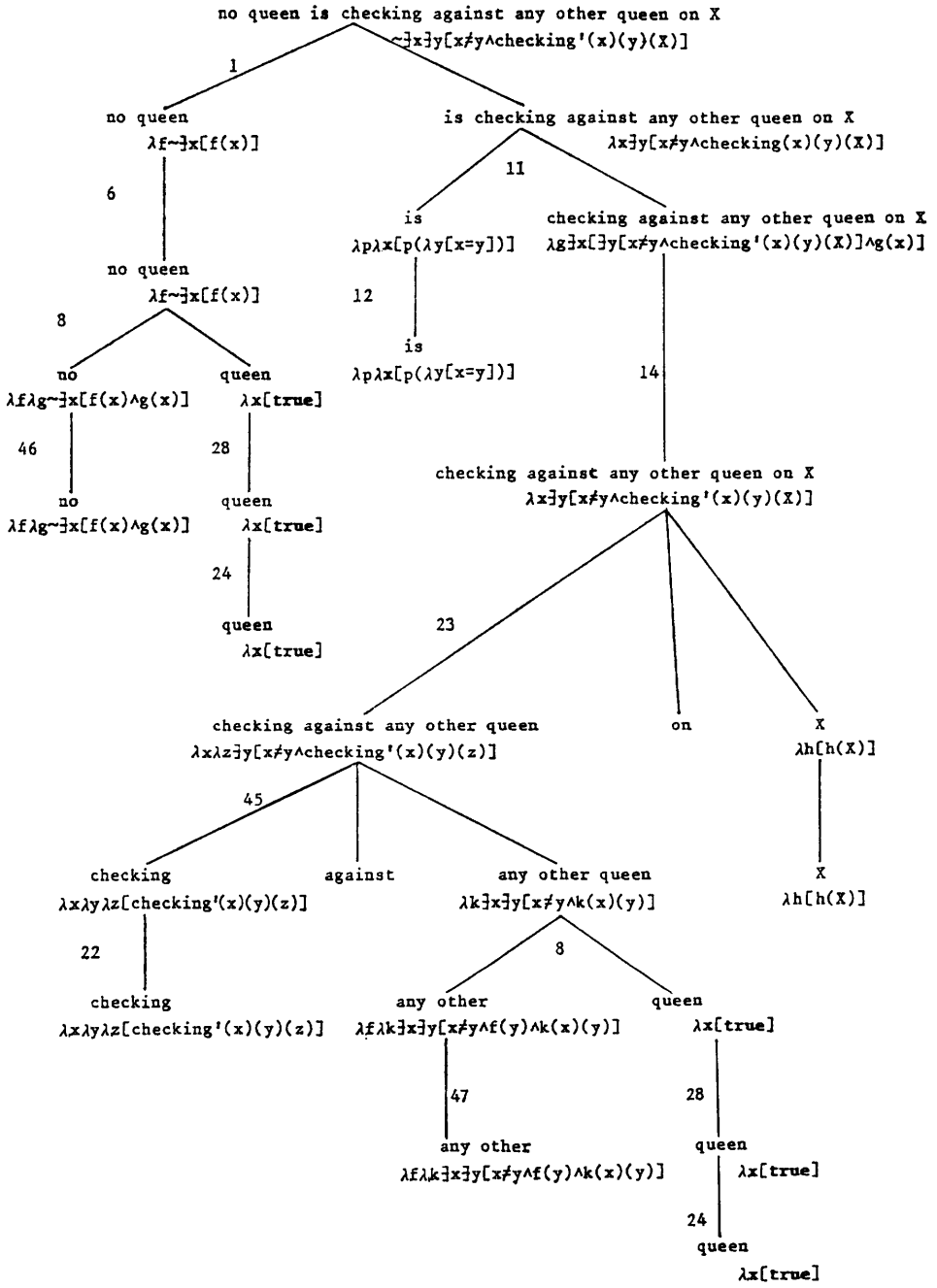


図 5 構文解析木
Fig. 5 Derivation tree.

参 考 文 献

1) Teichroew, D. and Herskey, E. A., III : PSL/PSA : A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems, *IEEE Trans. Softw. Eng.*, Vol. SE-3, No. 1, pp. 41-48 (1977).

2) Bell, T. E., Bixler, D. C. and Dyer, M. E. : An Executable Approach to Computer-Aided Software Requirements Engineering, *ibid.*, pp. 49-60 (1977).

3) Robinson, L. and Levitt, K. N. : Proof Techniques for Hierachically Structured Programs, *Comm. ACM*, Vol. 20, No. 4, pp. 271-283 (1977).

4) Ross, D. T.: Structured Analysis (SA): A Language for Communicating Ideas, *IEEE Trans. Softw. Eng.*, Vol. SE-3, No. 1, pp. 16-34 (1977).
5) Nakajima, R., Honda, H. and Nakahara, H.: Hierachical Program Specification and Verification—A Many-Sorted Logical Approach, *Acta Inf.*, Vol. 14, Fasc. 2, pp. 135-155 (1980).
6) Guttag, J. V., Horowitz, E. and Musser, D. R.: Abstract Data Types and Software Validation, *Comm. ACM*, Vol. 21, No. 12, pp. 1048-1064 (1978).
7) Futatsugi, K. and Okada, K.: Specification Writing as Construction of Hierarchically Structured Clusters, Proc. of IFIP Congress '80, pp. 287-292 (1980).
8) Montague, R.: *The Proper Treatment of Qualification in Ordinary English, Approaches to Natural Language*, Reidel Dordrecht, Holland (1973).
9) Hausser, R. and Zaefferer, D.: Question and Answering in a Context-Dependent Montague Grammer, in Guenthner, F. and Schmidt, S. J. (ed.), *Formal Semantics and Pragmatics for Natural Languages*, pp. 339-358, Reidel Dordrecht, Holland (1978).
10) McCarchy, J.: Circumscription—A Form of Non-Monotonic Reasoning, *Artif. Intell.* Vol. 13, No. 1, 2, pp. 27-39 (1980).
11) McDermott, D. and Doyle, J.: Non-Monotonic Logic 1, *ibid.*, pp. 41-72 (1980).
12) Pnueli, A.: The Temporal Logic of Programs, Proc. of the 18th Symposium on the Foundations of Computer Science, ACM (1977).
13) Friedman, J.: Expressing Logical Formulas in Natural Languages, in Groenendijk, J. A. G., Janssen, T. M. V. and Stokhof, M. B. J. (ed.), *Formal Methods in the Study of Languages*, pp. 113-130, Mathematisch Centrum, Holland (1981).

(昭和57年11月26日受付)
(昭和58年 9 月13日採録)

付録 構文規則と意味規則

(p. 215の図を参照ください)

付録　構文規則と意味規則

<sentence>::=<subject>$_\alpha$(P) <verb phrase>$_\alpha$(F)| :1
　　P(F)
　　for every name$_1$,...,name$_n$ which <sub sentence>$_\alpha$(F), <sentence>$_\alpha$(M)| :2
　　∀name'$_1$...∀name'$_n$[F(name'$_1$)∧...∧F(name'$_n$) → M]
　　<sentence>(M) <connective>(C) <sentecne>(L) :3
　　C(M)(L)
　　<there subject>$_\alpha$(A) <be verb>$_\alpha$(W) <subject>$_\alpha$(P) :42
　　A(W(P))

<subject>$_\alpha$::=<term>$_\alpha$(P) :6
　　P

<there subject>$_\alpha$::=there :43
　　λf∃x[f(x)]　　(positive)
　　λf∀x[f(x)]　　(negative)

<term>$_\alpha$::=name$_\alpha$| :7
　　λf[f(name'$_\alpha$)]
　　<determinar>$_\alpha$(D) <noun clause>$_\alpha$(F)| :8
　　D(F)
　　<plural>$_\alpha$(E)| :9
　　λf∃S[E(f)(S)]
　　<number>$_{integer}$(N) <plural>$_\alpha$(E) :10
　　λf∃S[number(S)=N∧E(f)(S)]

<term2>$_\alpha$::=<determinar2>$_\alpha$(D) <noun clause>$_\alpha$(F) :44
　　D(F)

<verb phrase>$_\alpha$::=<be verb>$_\alpha$(W) <complement>$_\alpha$(P) :11
　　W(P)

<be verb>$_\alpha$::=is (are) | :12
　　λpλx[p(λy[x=y])]
　　is not (are not) :13
　　λpλx[¬p(λy[x=y])]

<complement>$_\alpha$::=<adjective phrase>$_{[\alpha]}$(F)| :14
　　λg∃x[F(x)∧g(x)]
　　<term>$_\alpha$(P) :15
　　P

<adjective phrase>$_{[\alpha_1,...,\alpha_n]}$::=adjective$_{[\alpha_1,...,\alpha_n]}$| :22
　　λx$_1$...λx$_n$[adjective'(x$_1$)...(x$_n$)]
　　<adjective phrase>$_{[\alpha_1,...\alpha_{n+1},...,\alpha_n]}$(U) <preposition> <term>$_{\alpha_{n+1}}$(Q)| :23
　　λx$_1$...λx$_n$∃x$_{n+1}$[U(x$_1$)..(x$_{n+1}$)..(x$_n$)∧Q(λz[z=x$_{n+1}$])]
　　<adjective phrase>$_{[\alpha_1,...\alpha_{n+1},...,\alpha_n]}$(U) <preposition> <term2>$_{\alpha_{n+1}}$(Q) :45
　　λx$_1$...λx$_n$∃x$_{n+1}$[U(x$_1$)..(x$_{n+1}$)..(x$_n$)∧Q(λy$_1$λy$_{n+1}$[y$_1$=x$_1$∧y$_{n+1}$=x$_{n+1}$])]

<noun phrase>$_{[\alpha_1,...,\alpha_n]}$::=common noun$_{[\alpha_1,...,\alpha_n]}$| :24
　　λx$_1$...λx$_n$[common noun'(x$_1$)...(x$_n$)]
　　<adjective phrase>$_{[\alpha_1]}$(F$_1$) common noun$_{[\alpha_1,...,\alpha_n]}$| :25
　　λx$_1$...λx$_n$[common noun'(x$_1$)...(x$_n$)∧F$_1$(x$_1$)]
　　common noun$_{[\alpha_1,...,\alpha_n]}$ <adjective phrase>$_{[\alpha_1]}$(F$_1$)| :26
　　λx$_1$...λx$_n$[common noun'(x$_1$)...(x$_n$)∧F$_1$(x$_1$)]
　　<noun phrase>$_{[\alpha_1,...\alpha_{n+1},...,\alpha_n]}$(U) <preposition> <term>$_{\alpha_{n+1}}$(Q) :27
　　λx$_1$...λx$_n$∃x$_{n+1}$[U(x$_1$)..(x$_{n+1}$)..(x$_n$)∧Q(λz[z=x$_{n+1}$])]

<noun clause>$_\alpha$::=<noun phrase>$_{[\alpha]}$(F)| :28
　　F

<plural>$_\alpha$::=<noun clause>$_\alpha$-s(F) :33
　　λfλS∀x[element(x,S) ↔ F(x)∧f(x)]

<determinar>$_\alpha$::= a (the) | :34
　　λfλg∃x[f(x)∧g(x)]
　　every :35
　　λfλg∀x[f(x)→g(x)]
　　no :46
　　λfλg¬∃x[f(x)∧g(x)]

<determinar2>$_\alpha$::= any other :47
　　λfλb∃x∀y[x≠y∧f(y)→b(x)(y)]　(positive)
　　λfλb∃x∃y[x≠y∧f(y)∧b(x)(y)]　(negative)

タイプ
x,y: α , z: α$_{n+1}$,
x$_i$,y$_i$: α$_i$, F,f,G,g: <α t> ,
Q: <<α$_{n+1}$ t> t> ,
p,P: <<α t> t> ,
M,m,L,l: t , N: integer ,
W: <<<α t> t> <α t>> ,
U: <α$_1$<...<α$_{n+1}$<...<α$_n$ t>..> ,
E: <<α t> <set of α t>> ,
S: set of α , C: <t <t t>> ,
F$_1$: <α$_1$ t> , b: <α <α t>> ,
D: <<α t> <<α t> t>>