13. イオタ語の言語処理およびプログラム作成支援システム

京都大学
数理解析研究所
湯浅太一, 中島玲二
Taiichi Yuasa　Reiji Nakajima

§1. はじめに

ソフトウェアの信頼性向上やプログラム検証の困難さの軽減に，データ抽象化(data abstraction)や手続き抽象化に基づく，階層的でモジュラーなプログラミングが有効であることは広く認められている。またこの方法によるプログラム開発に適した環境を与える研究が行なわれている［例えば 6］。

一般にこれまでの研究は主にプログラム言語の設計が中心であったが，プログラム言語はより良い環境を与えるための一要素にすぎず，言語の目的とする方法論が有効に働くためには，その言語の埋め込まれる環境が重要である。特に階層的でモジュラーなプログラミングは，個々のモジュールと同時に，モジュール間の相互関係を重視するので，プログラム言語やプログラミング・システムを含めた総合的な考察が必要である。このような目的から，イオタ・システムとよばれる階層的でモジュラーなプログラミングを支援するための総合的なシステムが開発されてきた。本論では，このシステムを概観し，その考え方と方法を述べる。イオタ語とその数学的基礎，検証方法などについて詳しくは［9, 10］を見られたい。

§2. 言語

イオタ・システムはプログラムと形式的仕様（以下‘仕様’と略す）の記述のための言語，イオタ語を基礎としている。イオタ語で書かれるモジュールは，そのモジュールで定義される関数の宣言や仕様などを与える抽象的な部分（抽象部分，abstract part）とそのプログラム実現を記述する部分（実現部分，implementation part）が明確に区別して与えられる。あるモジュールMが他のモジュールの上に作成される時，Mはこれらのモジュールの抽象部分のみに依存し，それらの実現部分には全く依存しない。（以下モジュール間の‘依存関係’はこの意味で用いる。）

イオタ語は支援システムに埋めこまれることを仮定して設計されており，言語の機能をこえると思われるものはシステムにまかせることによって，言語自体は簡潔で統一のとれたものとなっている。例えば，各データ型の等号（equality）と代入（assignment）の実行が禁止されていないかどうかの管理がそうである。またインライン・コードなどによる最適化の指定もシステムへのコマンドによって行なわれる。イオタ語における仕様は，イオタ・システムに於いて，プログラムの開発と変更のためのツールとしての機能をもつ。モジュールの実現は，そのモジュールを procedural に記述したものであり，仕様は同じモジュールの declarative で抽象的な記述である。両者は互いに補間的な役割をもち，その記述を平行して行ない検証することにより，プログラムの信頼性が保証しやすいと考えられる。（階層的なプログラム開発においては，階層の下位に属するモジュールは，仕様も実現も多分に上位モジュールの実現のための便宜的存在であることが多いことに注意［12］。）　このような仕様の利用法は，イオタ語の仕様が部分的(partial)仕様であることに基づいている［11］。

§3. 各モジュールを独立に処理すること

階層的でモジュラーなプログラミングは，元来主としてプログラムの検証の困難を克服するために提案されたものであり，検証がモジュールごとに行なわれるのは当然であるが，プログラムのモジュール構造を有効に利用するためにはプログラムの入力，変更，デバッグ，コンパイルなどもモジュール単位に行なえることが望ましい。

モジュールごとのコンパイルには例えば次のような利点がある。第1に，あるモジュールの実現に変更があった場合，既にコンパイルされた他のモジュールは影響をうけないのでコンパイルしなおす必要がない。実行時間の短縮のためには，最終的に完成した一連のモジュールを一括してコンパイルすることも望ましいが（種々の最適化がこれによって可能となる。），プログラムの維持管理のためにはモジュールごとにコンパイルされたコードをプログラム・データベースに保つ必要がある。さらに，モジュールごとの構文解析や動的デバッグが有効に行なえる。

イオタ語によって書かれたモジュールはsype〔9〕の概念を用いてtype parameter化が可能であり，この目的のためには特殊なテクニックが必要である。簡単のためtrivialな例をあげる。いま2つの引数をうけとり，その大きい方をかえす関数'max'を作りたいとしよう。maxの引数の属するデータ型は，元（object）の間に全順序関係'≤'が定義されていれば何でもかまわない。この条件を記述するためにsype 'order'が作成される。（図I参照。ここで'@'はこのモジュールで定義されるsype即ちorderをあらわす。'AS'によって関数名の関数記号による省略が導入される。つまりless(x,y)と書くかわりにx≤yと書けることを意味する。）このsypeの上に，maxを定義するモジュールsort(p:order)がつくられる。（図II参照）ここでpはタイプパラメーターで，orderの条件をみたすtypeに束縛される。REALIZATION部には↓max（maxの実現関数）のbodyが与えられている。IF文の条件式'x≤y'に注目されたい。pがあるtype Tに束縛された時は，'≤'はTで定義された関数をあらわす。例えばpがtype integerに束縛されれば，integerの'≤'がよばれるであろう。　このように，この条件式でよばれる関数はpがあるtypeに束縛されてはじめて決定される。ではどのようにしてsort(p:order)を独立して（即ちpの束縛されるtypeに依存しないで）コンパイルできるだろうか？　この問に対する1つの解答はCLU〔1〕で与えられているが，イオタ語ではsypeを利用することによってより効率的な方法が開発されている。まずあるモジュールMで定義される関数のbodyでmaxがよばれ，このときpはあるtype Tに束縛されるとしよう。Mをコンパイルする時に，orderで定義された関数に対応するTの関数のテーブルを，それらの関数がorderのINTERFACE部にあらわれる順序に従って作成し（orderの場合関数が≤しかないので単純だが一般のsypeは複数個含む），このテーブルを実行時にmaxにおくる。↓maxのIF文の条件式では，やはりorderのINTERFACE部を調べることによって，このテーブルから実際によばれる関数をとり出してよぶようにコードが作成されている。

一般の場合は，タイプパラメーター機能に伴う問題はこの例のように単純ではなく，本質的な困難さを含む。一般的な問題点とその解決の方法については〔13〕に詳しく述べられているので参照されたい。

```
INTERFACE SYPE order

        FN less: (@,@) -> @   AS @=<@

END INTERFACE


SPECIFICATION SYPE order

        VAR x,y,z:@
        AXIOM    1. x=<y V y=<x
                 2. ( x=<y & y=<z ) => x=<z
                 3. ( x=<y & y=<x ) => x=y

END SPECIFICATION
```

図 I

```
INTERFACE PROCEDURE sort(p:order)

        FN max: (p,p) -> p

END INTERFACE


REALIZATION PROCEDURE sort(p:order)

        FN ^max (x,y:p) RETURN (z:p)
                IF x=<y THEN z:=y
                        ELSE z:=x
                ENDIF
        ENDFN

END REALIZATION
```

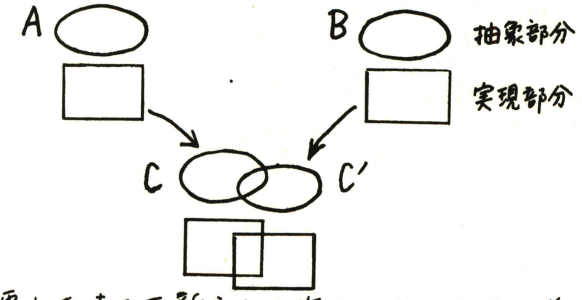
(ここで ^max は"DEC上の↓maxの表示, =< は≦, V はOR, & はAND, => は"ならば"である。)

図 II

イオタ語のコンパイルに関するもう一つの重要な問題は，代入文や関数呼び出しなどの際の値のうけわたしである。一般に(抽象)データ型をユーザーが定義できる言語においては、arrayやrecordなどの組合せにより，個々の値が物理的に大きく得る。例えばイオタ語では代入文 x:=y を実行する場合、意味論上はyの値を複製(copying)することになっているが、実際のコードではできる限り意味論上の正しさを保ちながら，不必要な複製を制限するように最適化されている。これに関しては[14]が詳しい。

§4. モジュールとモジュール間の関係の管理

一度作成されたモジュールは絶えず修正の可能性を含んでいる。例えば右図において、モジュールAは検証までCの仕様を用いて完了しているとする。このときCの上にBを書きたいが、Bの要求がAとやや異なる場合、Cをかえることによ ってAに費した努力がそこなわれることになりかねない。このためCを若干変更したものを新らしく作り，Cとともに並

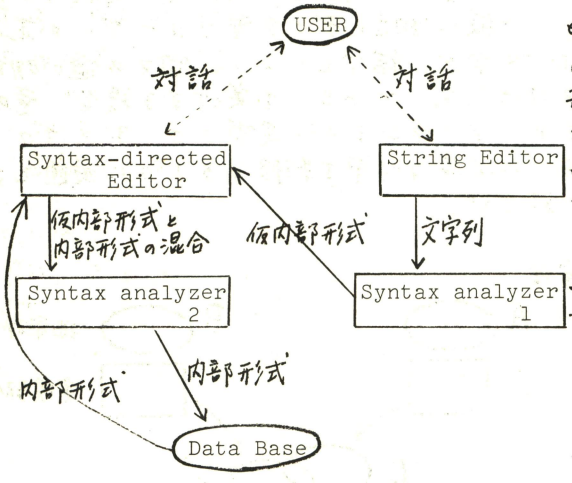


存させると都合がよい。一般に同じモジュールの多くのバリエーションを仮想的に維持することが便利であり、イオタ・システムにとり入れられている[11]。(前述の、イオタにおける仕様のあり方と関係していることに注意されたい。)

このような目的に加え、モジュール間の関係がcircularになることを防止するなどのために、システムは常にモジュール間の関係を管理する必要がある。この管理はすでに完成したモジュールについてのみでなく、未完成のものも含む。

§5. モジュールの入力と変更

システムがプログラムの作成を支援するためにはユーザーとの対話が非常に望ましい。対話を効果的に行なうためには、システム全体に共通したプログラムの内部表現を持ち、各サブシステムがこれを直接扱える必要がある。LISPのように、内部表現とプログラム・テキストの対応が明らかな場合はこれは容易だが、ALGOL型の言語では、内部表現への変換には構文解析を必要とし、この目的のためにはプログラムの入力と変更に特別な工夫が必要である。特にイオタ語の場合、モジュールの構文解析は、それが依存する他のモジュールの抽象部分にある情報を用いて処理することが必要であり、かなりの時間を要する。例えば、関数はその関数を定義するモジュールの名前と関数名によって一意的に決定されるが、プログラムの中の式(expression)では、関数名だけ、あるいはinfixまたはprefixの関数記号による省略(AS abbreviation)も可能である。同一の関数名や関数記号が二つ以上のモジュールで導入されている場合は、式全体のタイプチェックによって関数を識別する必要がある。従って、プログラムの変更の際には、一度作成された内部表現を有効に利用し、重複した処理を避けることが望ましい。このような理由から、従来のscannerやparserの単純な組合せによるschemaでは不十分で、次のようなSyntax analyzer, Syntax-directed editor, String editorの有機的な結合が効果的である。



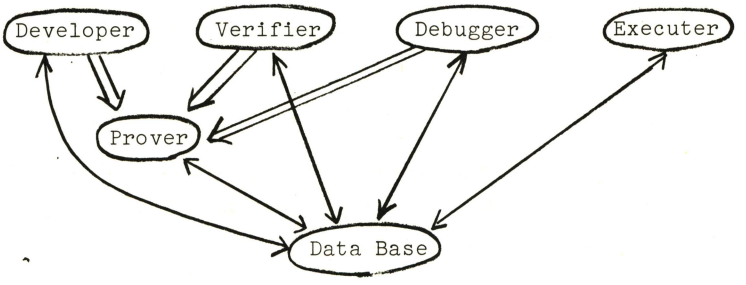
新らしいソースプログラムはString editorを介してSyntax analyzer 1に与えられ、大まかな構文木(仮内部形式)が作成される。ここでエラーが検出されれば自動的に再度String editorがよばれ、(エラーの発見された部分にポインターがセットされて)エラーの局所的な修正が行なわれる。String editorにはreserved wordやユーザーの与えたidentifierを自動的に展開する機能やプリティプリンターなどが備わっている。

Syntax analyzer 1で作成された仮内部形式はSyntax analyzer 2によって、データ型の分析、式にあらわれる関数の識別などが行なわれ、内部形式に変換されると同時にモジュール間の関係に関する情報がとり出される。

ここでエラーがあれば，Syntax-directed editorを通じて修正される。　この際の入力は再度String editor，Syntax analyzer 1を通して行なわれ，修正部分が新たな仮内部形式におきかえられる。　この結果，Syntax analyzer 2がすでに処理した内部形式の中に仮内部形式が混在したものができる。　これを再度Syntax analyzer 2が処理するのであるが、仮内部形式の部分のみを処理し，すでに処理ずみの部分はそのまま保存するように設計されている。　これはイオタにおける構文解析が時間がかかることから大いに有効である。

§6. イオタ・システムの概観

イオタ・システムはDeveloper，Verifier，Debugger，Executer，Proverの五つのサブシステムから構成され，これらは階層的でモジュラーなプログラム作成に必要なすべての情報をもったData Baseによって互いに密接に関連している。（下図参照。二重線はコントロールの流れ、実線はデータの流れをあらわす）



Developerはユーザーとの対話によって，イオタ語でユーザーが作成するモジュールの入力と変更を行ない，内部形式のプログラムをモジュールとモジュール間の関係に関する情報とともにData Baseに保存する。

Verifierは　各モジュールの仕様が、そのプログラムの実現によって満たされていることを対話的に検証する。Proverはmany-sortedな一階述語論理（イオタ論理［9］）のための対話的定理証明システムであり、特に階層的かつモジュラーな構造をもった理論の上の演繹を能率的に行なうように設計されている［4］.（Proverは Verifier から呼ばれる以外に、モジュール間の関係（sype-type関係の意味論的チェック）を解析するためにDeveloperからもよばれる。また，モジュールの仕様を活用したシンボリックなデバッグにも使える。）

Debuggerは、Data Base内の内部形式プログラムを直接実行するインタプリタとtracerやbreakerなどの（通常の意味の）デバッグ機能からなり、動的なデバッグを支援する。Executerは、Developerによって生成された内部形式から、オブジェクト・コードを生成、実行するものである。

あとがき

イオタ・システムは現在、京都大学数理解析研究所において、DECシステム20の上で開発がすすめられている。同時にIBM 370と日立及び富士通のMシリーズへ移植される。システムの記述にはLISPが適しているが、どのLISPシステムが適当かについては問題があった。結局、イオタ・システムが大規模なものであることから、サポートは少ないが、大きなユーザー・プログラムを効率よく走らせるLISPとしてUtah LISP[7]が採用された。しかし、デバッグやLISP向きのエディタなどの強力な支援システムが必要なので、Utah LISPと同様、Stanford LISP 1.6系統のUCILISP[2]を使って開発が行なわれている。これには複数のTSSジョブを同時に同一端末から走らせるDEC 20のOS, TOPS-20のPTYCON[3]が便利に応用できる。(即ち、同一のプログラムを2つのLISPシステムの上で同時に走らせてテストしている。) プログラム開発のための強力な支援システムを持ち(例えば、interlispのように), 開発後は実行に必要な部分のみを独立して効率的に使えるLISPシステムが望ましいが、DEC 10, 20の上でも見あたらなかった。将来のLISPシステムの設計にはこの点を考慮に入れていただきたい。

謝辞

イオタ・プロジェクト(特にProver)には香川大学の本田道夫氏も参画している。また京都大学の小島啓二氏の協力もうけている。名古屋大学の金田康正氏には、Utah LISPについて有益な情報をいただいた。

参考文献

1. Atkinson, R., Liskov, B., Scheifler, R.: Aspects of implementing CLU, M.I.T. Computer Structures Group Mero 167, 1978.

2. Bobrow, R. J. et al.: UCI LISP MANUAL.

3. DECSYSTEM - 20 Operator's Guide, 1978.

4. Honda, M., Nakajima, R.: An interactive theorem proving system for structured theories, IJCAI '79.

5. Huet, G. et al.: Environnement de programmation Pascal, Mannual d' utilization. sous siris 7/8. IRIA, 1977.

6. Liskov, B., et al.: Abstraction mechanism in CLU, CACM 8 567-576, 1977.

7. Mavti, J. B. et al.: STANDARD LISP REPORT, Univ. of Utah, 1978.

8. Nakajima, R., Honda, M., & Nakahara, H.: Describing and verifying programs with abstract data types, in Formal Description of Programing Concept (ed. Neuhold) North-Holland Publishing Co. 1977.

9. Nakajima, R., Nakahara, H., Gonda, M.: Hiercerchical program specification and Verification — a Many-sorted logical approach —, RIMS-Preprint 265, 1978

10. 中島玲二、中原早生、本田道夫：イオタ語とイオタ・システム—階層的プログ

ラム開発とその検証のための言語と方法とシステム，第19回プログラミング・シンポジウム，1978.

11. Nakajima, R., Yuasa, T.: Program development with the ι system,RIMS-Preprint, 1979.

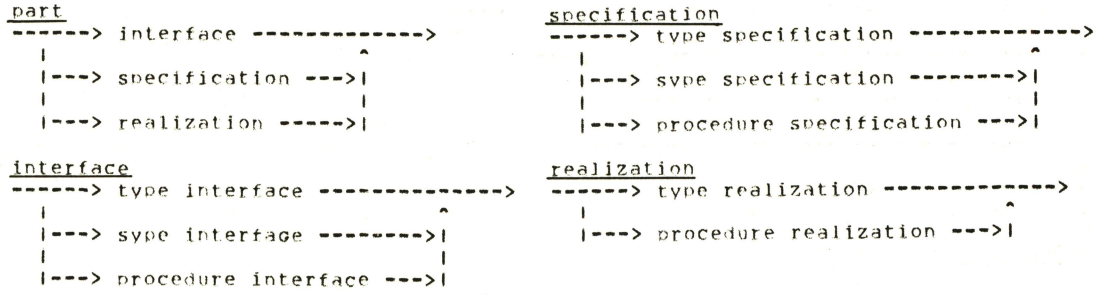
12. Parnas, D. L.: The use of precise specification in the development of software, IFIP 77.

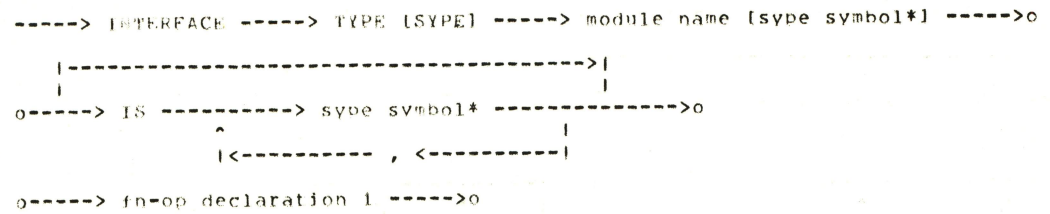
13. Yuasa, T.; Module-wise compilation for a language with type-parameterization mechanism,RIMS Preprint-280 1979.

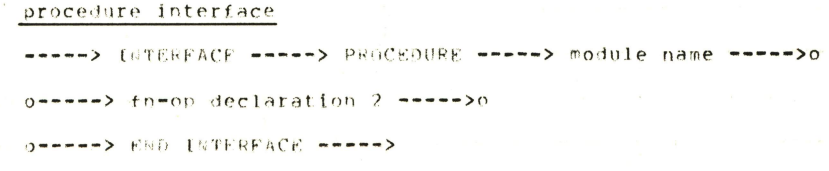
14. Yuasa, T.: Design and implementation of ι language system,RIMS-Preprint 1980.

付録 イオタ語の構文

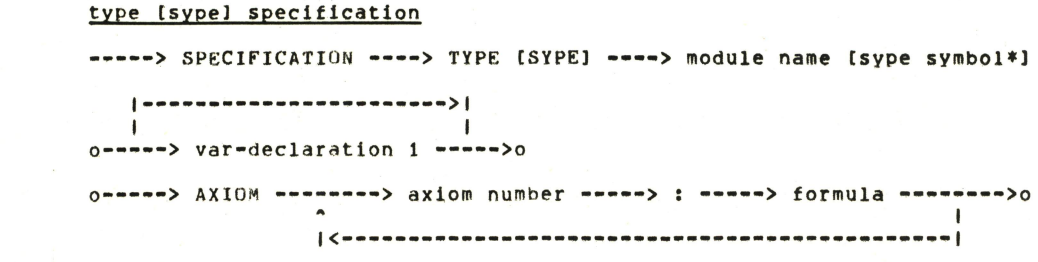
イオタ語のシンタックスを与える．但しexpressionとformulaについては略す．arrayやrecordのようなbuilt-in type, sypeなどについても略す。現在イオタ語のreference manualが作成中であり、詳細はそれを参照されたい。



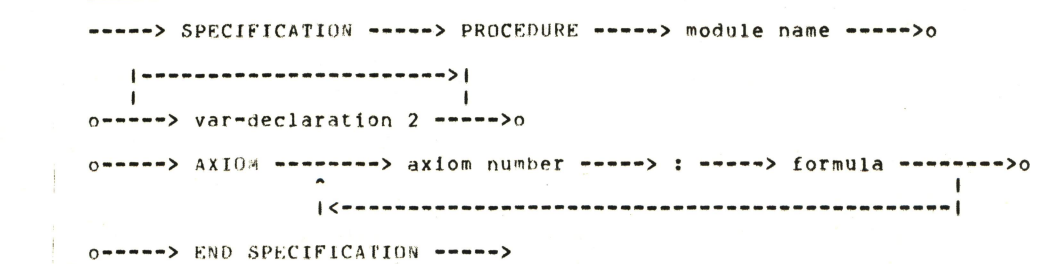




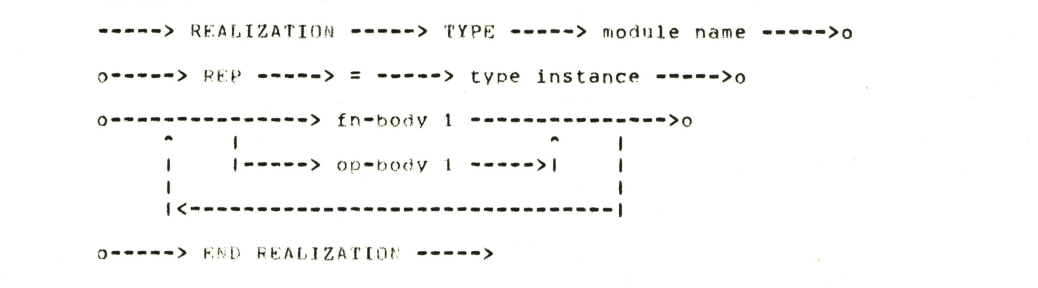
type [sype] specification



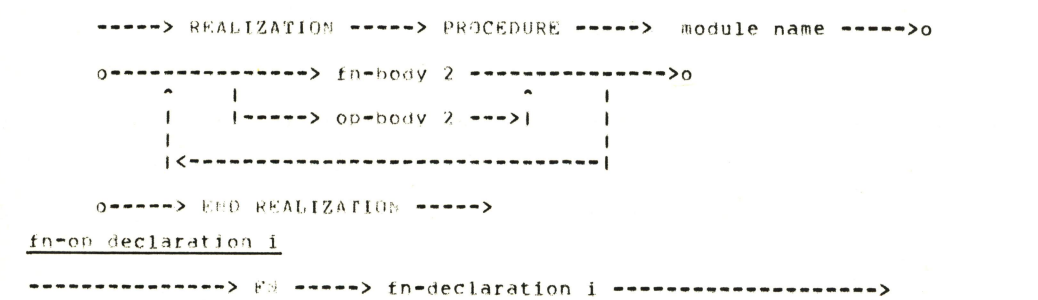
procedure specification



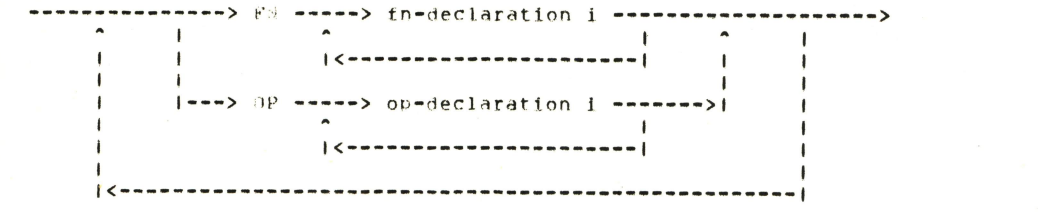
type realization



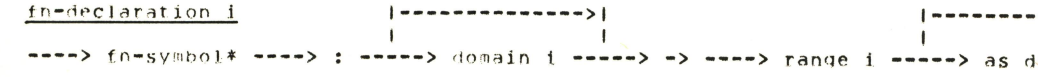
procedure realization



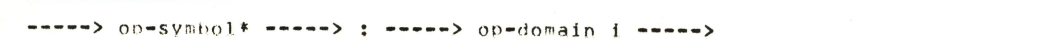
fn-op declaration i



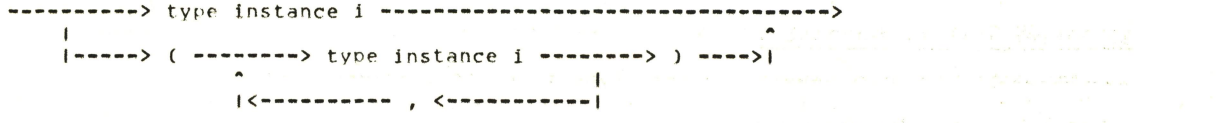
fn-declaration i



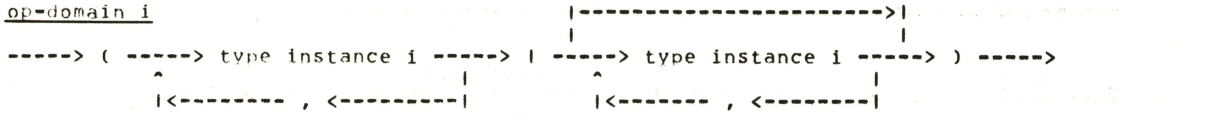
op-declaration i



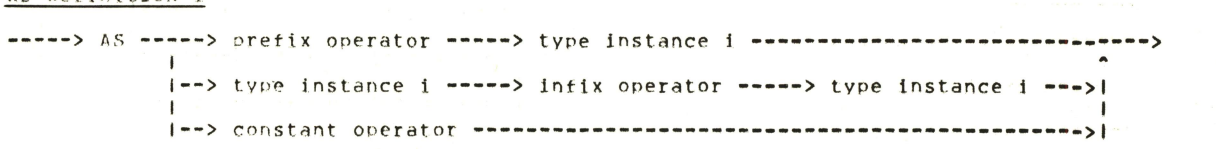
domain i, range i



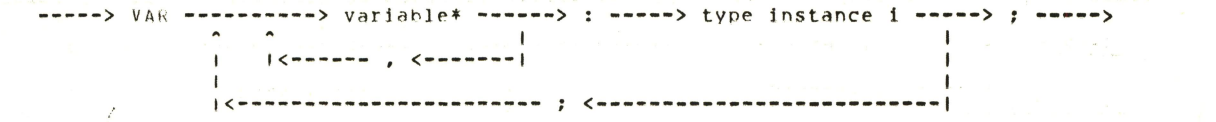
op-domain i



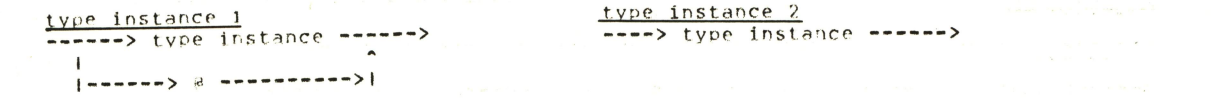
as-definition i



var-declaration i

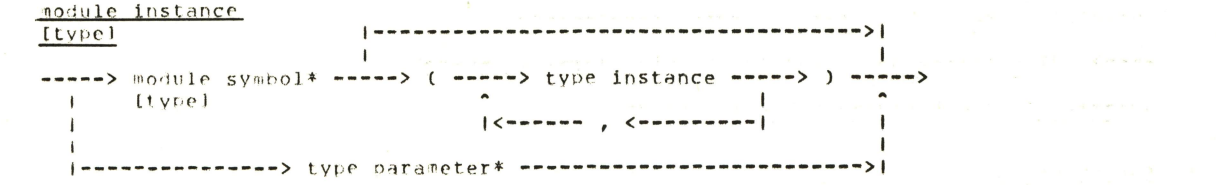


type instance 1

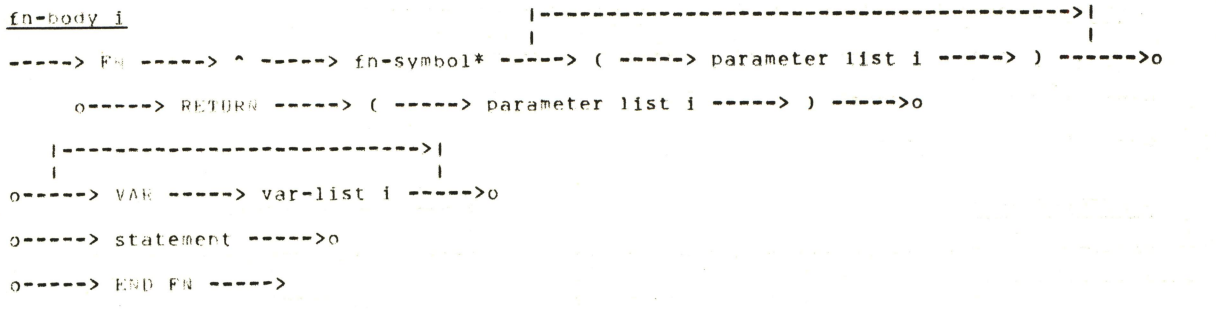


type instance 2

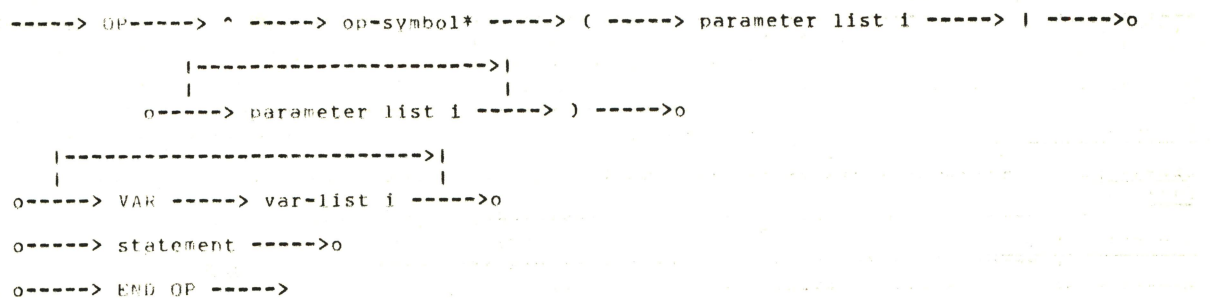
module instance
[type]



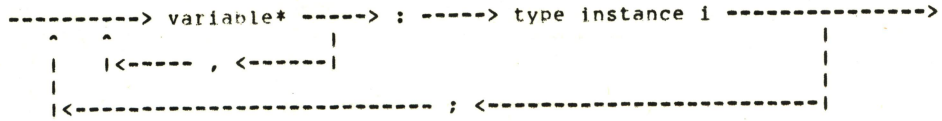
fn-body i



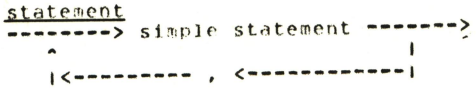
op-body i



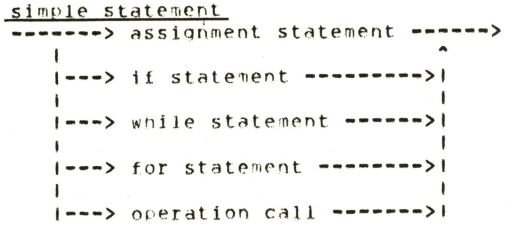
parameter list i, var-list i



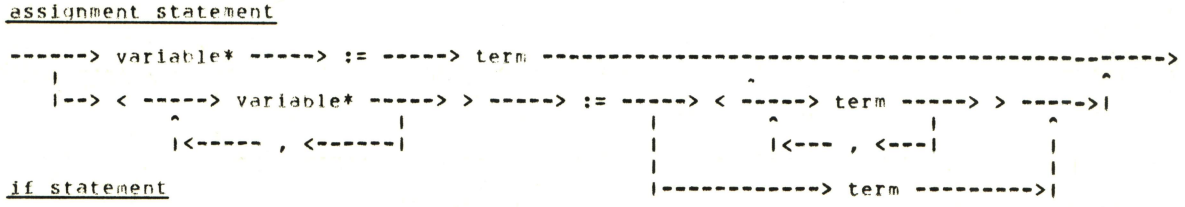
statement



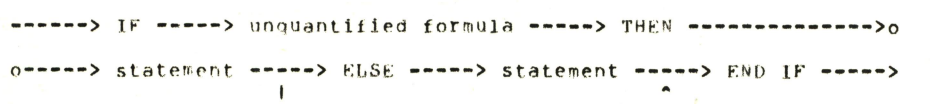
simple statement



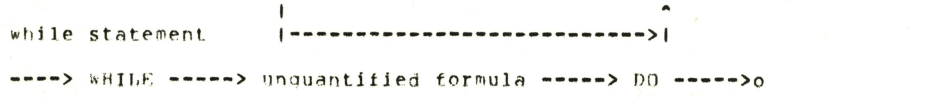
assignment statement



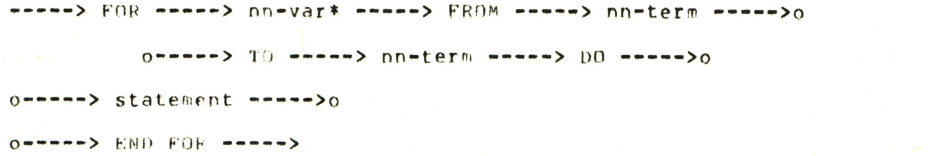
if statement



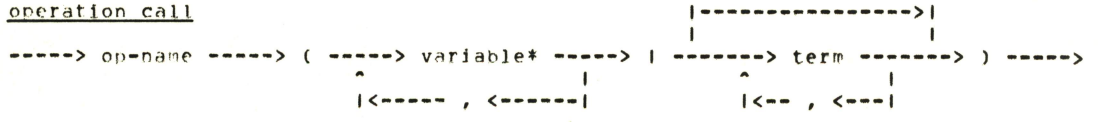
while statement



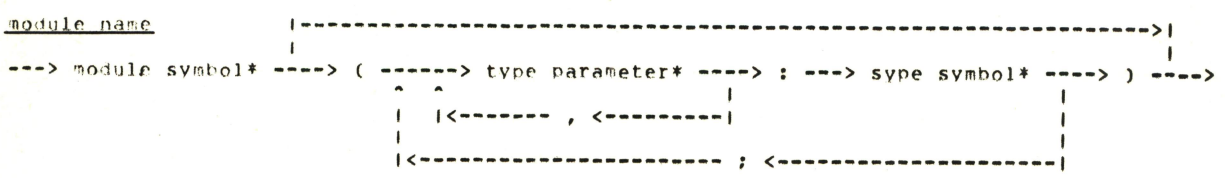
for statement



operation call



module name



fn-name
[op]