ソフトウェア工学　52 — 2
（１９８７・２・１３）

拡張型統合化プログラミング環境の構築

天満　隆夫[†]，　吉野　真澄[†]，　坪谷　英昭[†]，　田中　稔[††]，　市川　忠男[††]
[†]広島大学大学院　[††]広島大学工学部

本稿では、拡張型統合化プログラミング環境ＭＵＬＳについて述べる．ＭＵＬＳは、(1)環境が保持するプログラミング言語及びツールに関する情報を取り換えることにより、各言語に対し個別の専用化された機能、ならびに(2)言語やツールの追加や変更を支援する機能、を提供する．プログラムの内部表現である属性付き抽象構文木の各ノードやツールをオブジェクトとみなし、環境内の処理はオブジェクト間のメッセージの授受によって行なわれる．Ａｄａプログラムの編集、構成管理、デバッグを支援する機能を実現しＭＵＬＳの有効性を確認した．

Construction of An Extensible Integrated Programming Environment

Takao TENMA, Masumi YOSHINO, Hideaki TSUBOTANI, Minoru TANAKA, and Tadao ICHIKAWA
Faculty of Engineering, Hiroshima University
Shitami, Saijo-cho, Higashi-hiroshima, 724 Japan

In this paper, we present an extensible integrated programming environment MULS. MULS provides (1) language specific facilities by exchanging information on programming language and tool, and (2) facilities of addition and modification of languages and tools. Tools and nodes in an attributed abstract syntax tree which represents a program are regarded as objects, and all the operations in MULS are realized by message passings between objects. We implemented an Ada programming environment which supports editing and debugging a program and configuration management.

1. はじめに

プログラム開発の生産性・信頼性向上を支援するため多くの統合化プログラミング環境（Integrated Programming Environment）が開発されている[1],[2],[3]．統合化プログラミング環境とは、

(1) ツールが扱うプログラムの表現形式の統一化、
(2) ツールのユーザインタフェースの統一化、

によってツールが密に結合されているプログラミング・システムのことである[4]．一般に、統合化プログラミング環境は、

(1) 特定の言語向きの機能の提供、
(2) プログラムの持つ構造を利用した機能の提供、
(3) インクリメンタルなプログラム開発の支援、
(4) 対話的なプログラム開発の支援、

といった特徴を持つ．

統合化プログラミング環境を用いることにより、ユーザはプログラムの構造を利用した編集、インクリメンタルな実行・デバッグといった操作をモードの切り換えなしに行なうことができる．この操作の過程において、プログラムの構文・静的意味チェックなどが自動的に行なわれる．

しかしながら、統合化プログラミング環境は特定のプログラミング言語に強く依存するため、プログラミング言語ごとに個別のプログラミング環境を構築しなければならないといった問題がある．統合化プログラミング環境構築の負荷を軽減させるため、プログラミング言語の仕様からその言語向きのツールや環境を生成する研究が行なわれている[5],[6],[7]．以上の観点に基づき、筆者らは既にプログラミング言語の構文を利用した編集と編集時に構文・静的意味チェックを行なう言語指向エディタを、生成するシステムを開発している[8]．

本稿では、特定の言語に対する統合化プログラミング環境構築の支援と環境の機能の拡張・変更の容易化を目的とした、拡張型統合化プログラミング環境ＭＵＬＳについて述べる．ＭＵＬＳは次の特徴を持つ．

(1) 複数の言語の各々に対し専用化されたプログラミング機能を提供する．
(2) 言語やツールの追加・変更を支援する機能を提供する．
(3) プログラムの内部表現である属性付き抽象構文木のノード、およびツールをオブジェクトとみなし、ユーザの操作に対する環境内部の処理をオブジェクト間のメッセージの授受で行なう．
(4) 単にプログラムだけではなく、木構造を持つデータをプログラムと同様に扱えるので、バージョン木を扱うバージョン管理、プログラムのコンフィギュレーション木を扱う構成管理といった機能を、プログラムに対する機能の実現と同様の手法を用いて実現することができる．

実際に、Ａｄａプログラムの編集、デバッグ、構成管理を支援する機能を持つプログラミング環境を実現し、ＭＵＬＳの有効性を確認した．

2. 拡張型統合化プログラミング環境ＭＵＬＳ

2.1 ＭＵＬＳのシステム構成

図１にＭＵＬＳのシステム構成を示す．ＭＵＬＳは各言語に対してプログラミング機能を提供するためのＬＤＥ（Language-Dependent Environment）、プログラムおよびプログラムに固有の情報を保持するデータベース（program database）、言語に関する情報のためのデータベース（language database），ツールに関する情報のためのデータベース（tool database）からなる．

ＬＤＥは操作対象となるプログラムおよびプログラム固有の情報を保持するＰＳ（Program Space）、言語に関する情報（以下、Ｌ情報と呼ぶ）を保持するＬＩＳ（Language Information Space）、ツールに関する情報（以下、Ｔ情報と呼ぶ）を保持するＴＩＳ（Tool Information Space）および特定の言語に依存しない部分（図１には表示されていない）からなる．ＬＤＥの詳細は３章で述べる．

ＭＵＬＳはＬ情報・Ｔ情報を取り換えることによって、複数の言語の各々に対する機能を提供する．この取り換えは、ＬＤＥが扱うプログラムごとに行なわれる．

Ａｄａプログラムに対する操作を行なう場合を例に用いてＬ情報・Ｔ情報がＬＤＥにどのようにして取り込まれるかを説明する（図２）．図中、点線はデータベース内の情報の関係を表す．ユーザがあるＡｄａプログラムを指定すると、そのプログラムがまずプログラム用データベースからＬＤＥのＰＳに取り込まれる．次にそのプログラムが書かれた言語、すなわちＡｄａに関するＬ情報が言語用データベースからＬＩＳに取り込まれ、最後に言語ごとにその使用が定められているいくつかのツール（プリティプリンタやコードジェネレータなど）に関するＴ情報がツール用データベースからＬＤＥのＴＩＳに取り込まれる．

プログラムに対する操作を終了した時は、プログラムおよびプログラム固有の情報がプログラム用データベースに書き込まれ、ＬＤＥに別のプログラム、Ｌ情報、Ｔ情報を取り込むことが可能となる．

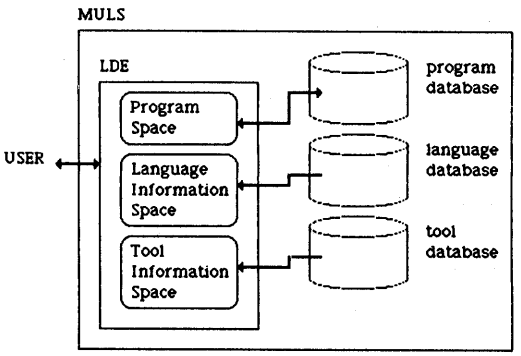

Fig.1 Configuration of MULS

２．２　Ｌ情報・Ｔ情報の記述

Ｌ情報・Ｔ情報は各々Ｌ記述（Language Description）・Ｔ記述（Tool Description）から作られる．ＭＵＬＳでは、これらの記述を行なうための言語（Ｄ言語；５章で説明する）を用意している．Ｄ言語用のＬＤＥを用いて、言語・ツールの追加・変更をプログラムの作成・修正と同様に行なうことができる．

ＡｄａのＬ記述を変更する場合の例を図３に示す．ＰＳにはプログラム用データベースからではなく言語用データベースから、Ａｄａに関するＬ記述が取り込まれる．また、ＬＩＳにはＤ言語のＬ情報が、ＴＩＳにはＬ記述からＬ情報を生成するトランスレータ（translator）を含めたＴ情報が、図２の場合と同様に取り入れられる．これによりＤ言語用のＬＤＥができ、Ｌ記述を変更できる．Ａｄａに関する記述の変更の終了後、ＡｄａのＬ記述とトランスレータによって生成されたＡｄａのＬ情報が言語用データベースに書き込まれる．

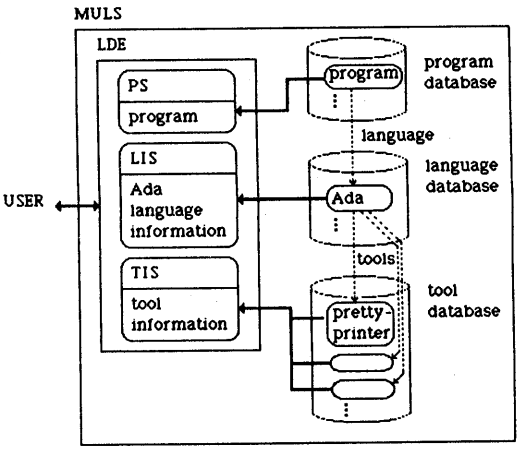

Fig. 2 A LDE for Ada

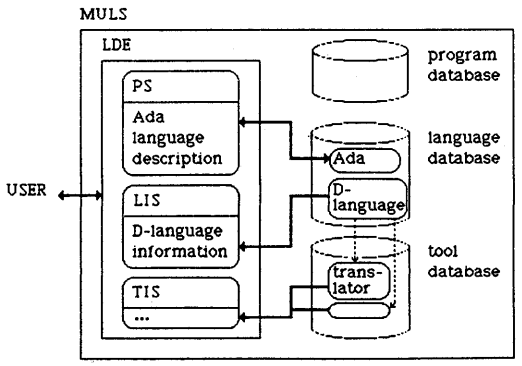

Fig.3 Modification of Ada Language Description

３．　ＬＤＥ内の処理メカニズム

３．１　オブジェクト指向の導入

異なる言語間でのツールの共有、Ｌ記述の作成・変更の容易化のために、オブジェクト指向プログラミング[9]の考え方をＬＤＥに導入した．ＬＤＥは、ツールならびにプログラムの内部表現である属性付き抽象構文木（Attributed Abstract Syntax Tree；以下、構文木と呼ぶ）の各ノードをオブジェクトとみなし、ユーザの操作に対するＬＤＥ内部の処理をオブジェクト間のメッセージの授受によって行なう．また、Ｌ記述・Ｔ記述の容易化のためクラス間に階層関係を導入した．

Ｌ情報、Ｔ情報はそれぞれＬ記述、Ｔ記述から生成される．これらの情報はいくつかのクラスよりなる．クラスはオブジェクトがメッセージを受け取ったときの動作（メソッド）、オブジェクト間の関係（リンク）、オブジェクトの持つ変数（属性）、上位クラスなどの情報を保持している．

３．２　ＬＤＥの構成

図４にＬＤＥの構成を示す．ＬＤＥは、

(1) Ｌ情報を保持するＬＩＳ、
(2) Ｔ情報を保持するＴＩＳ、
(3) プログラムの内部表現である構文木（ＡＳＴ）、ツールを表すいくつかのツール・オブジェクト、コンテキスト・オブジェクトを保持するＰＳ、
(4) 特定の言語に依存しない部分（図４には表示されていない）、

からなる．

(3)のコンテキスト・オブジェクトはＬ情報、Ｔ情報、およびＬＤＥ内のすべてのオブジェクトの管理ならびにコマンドの初期設定を行なうためのオブジェクトである．コンテキスト・オブジェクトはＬ情報、Ｔ情報、ＰＳが保持する情報のデータベースからＬＤＥへの取り込みおよびＬＤＥからデータベースへの書きだしをおこなう．また、

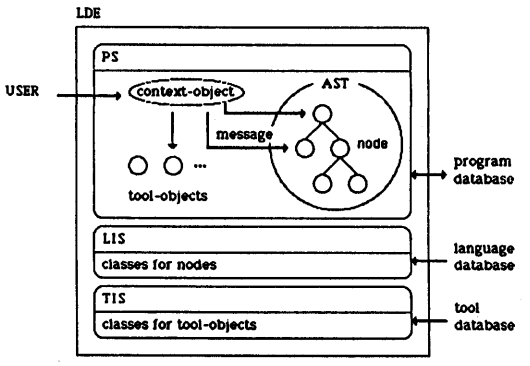

Fig. 4 Configuration of LDE

ＭＵＬＳでは言語、ツール、プログラムに対して使用可能なコマンドを定義することができる．これらのコマンドは、コンテキスト・オブジェクトのメソッドとして実現されている．

ユーザのコマンドはコマンド・インタプリタによってメッセージに変換された後、コンテキスト・オブジェクトに送られる。次にコンテキスト・オブジェクトから画面上のカーソル位置に対応する構文木内のノード、ルート・ノード、あるいはツール・オブジェクトへメッセージが送られる．その後ノード、ツール、コンテキストといったオブジェクト間のメッセージの授受によって処理が行なわれる．

３．３　クラス

Ｌ情報およびＴ情報はいくつかのクラスに関する情報を保持している．各クラスは、言語・ツールに関する以下のような情報を保持する．

(1) クラス名

クラスの名前．

(2) 上位クラス名

クラスは複数の上位クラス（super class）を持ち、属性、リンク、メソッド、条件付きメソッドを継承する．継承されたメソッドはオーバーライド（override）することができる．また、あるクラスに対し、そのクラス自身とそのクラスを上位クラスとするクラスをサブクラス（sub-class）と呼ぶ．

(3) リンク型

リンクはオブジェクトを値とする属性であり、オブジェクト間の関係を表すために、またメッセージを送るために用いられる．視覚的にはリンクはオブジェクトを結ぶ有向枝として表現される．リンクは主に構文木のノード間の構造的・意味的な関係を表すために用いる．

クラスはリンクの性質を決定する複数のリンク型を持つ．リンク型はリンク名、リンククラス、リンク数制約から定義される．リンククラスはクラスの名前からなり、結合されるオブジェクトのクラスがリンククラスのサブクラスでなければならない、といった制約を与える．この制約と継承によりリンクによって結合されるオブジェクトがリンククラスの持っている性質を持つことが保証される．リンク数制約とは、ひとつのオブジェクトが持つ同じリンク型のリンクの数の上下限値からなる．また、同じ型を持つリンクの間には順序関係があり、個々のリンクはリンク型名とリンクの順序を表す自然数でアクセスされる．

リンクの例を図５に示す．クラスＣのインスタンスａはリンク型Ｌ１のリンクＬ１．１、Ｌ２のリンクＬ２．１、Ｌ２．２を持っている．この場合、ｂのクラスはリンククラスＬＣ１のサブクラス、ｃ、ｄのクラスはＬＣ２のサブクラスでなければならない．また、ａに実際に結合されているリンク型Ｌ１とＬ２のリンク数は、それぞれ１と２なので共にリンク数制約 1..1 と 1..10 を満足している．

リンクには木構造を表すための構造リンクと、その他の関係を表す意味リンクがある．

(4) 属性

オブジェクトが持つインスタンス変数．

(5) メソッド

オブジェクトがメッセージをうけとったとき起動される手続き．メソッド名と手続き本体からなる．

(6) 条件付きメソッド

リンクによって結合されているオブジェクト、または自分自身のリンク・属性の値が変更されたとき、自動的に起動される無名のメソッド．インクリメンタルな静的意味チェックの記述を容易にするために用いる．

while に対するクラス定義の例を図６に示す．

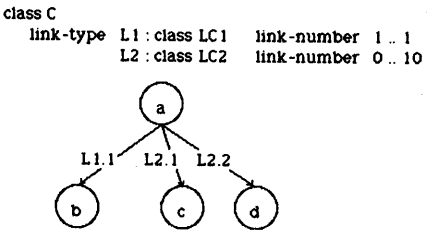

Fig. 5 Example of Link

```
class  while
       super          loop
       struct-link    condition     : expression  1 .. 1
                      body          : inherited
       semantic-link
       attribute
       method
              print  ()
                     (print 0 "while ")
                     (struct c condition)
                     (string c " loop")
                     (struct-para 1 body)
                     (string 0 "end loop ;")
              ...
       conditional-method
              condition.type
                     (if     (not (equal $condition.type
                                         'BOOLEAN))
                             'ERROR)
              ...
end
```

Fig. 6 "while" Class Definition

３．４　構文の表現と編集操作

ＭＵＬＳでは言語の構文（厳密には抽象構文）を、上位クラスと構文リンク型の定義で表す．例えば、

```
statement ::= if | loop | assignment | case
loop      ::= while | for
```

といったｏｒオペレータからなる構文規則は、右辺の構文要素が左辺の構文要素の一種であり、より具体化が進んだ構文要素であることを示していると考えられる．すなわち、左辺の構文要素は右辺の構文要素の上位クラスであり、上位クラスから下位クラスへ性質が継承されると考えられる．この例では、statement は if、loop、assignment、case や while、for などの上位クラスとなる．

一方、

```
if ::= expression                条件
       statement { statement }   ＴＨＥＮ部
       { statement }             ＥＬＳＥ部
```

といった構文規則は、if がひとつの expression（条件）、１個以上の statement（ＴＨＥＮ部）、０個以上の statement（ＥＬＳＥ部）から構成されることを表している．このような構文規則は、(1)リンククラス expression、リンク数制約 1..1 からなる condition、(2)リンククラス statement、リンク数制約 1..無限大 からなる then-part、(3)リンククラス statement、リンク数制約 0..無限大からなる else-part といったクラス if の３つの構造リンク（struct link）型の定義で表される（図７）．

以上の構文の表現に基づき、ＭＵＬＳではプログラムの構文的な正しさを常に保証するため、プログラムの編集にはテンプレートを用いる．編集操作には以下のようなものがある．

(1) ノードの置き換え

ノードをサブクラスまたはスーパークラスのノードで置き換える操作であり、それぞれテンプレートの挿入・削除に相当する．置き換わるノードにはリンク数制約の最小数だけリンククラスのノードが生成されて結合される．置き換えによる情報の損失を防ぐため、継承関係にある属性・リンクの値は置き換えられるノードから、置き換わるノードへ移される．loop を while に置き換える例を図８に示す．この例では while の condition リンク型を満足する１つの expression ノードが生成され結合されている．また置き換えられるノードのクラス loop から継承したリンク型 body のリンクに関しては、もとのノードの持つ２つのリンクが移されている．

(2) リンクの追加・挿入

リンクとそのリンクに結合されるノードを生成する操作．while ノードの body.1 リンクの次にリンクを追加する例を図９に示す．body のリンククラス statement のノードが生成され結合されている．またリンク数は３となるのでリンク数制約 1..無限大を満足している．

(3) リンクの削除

(2)の逆の操作．

link-type	:	condition	then-part	else-part
class	:	expression	statement	statement
number	:	1 .. 1	1 .. ∞	0 .. ∞

Fig. 7 Struct Links of "if" Class

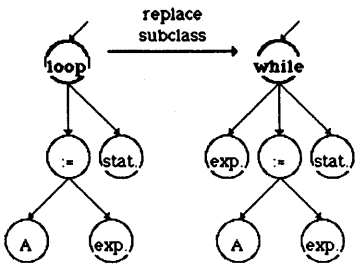

Fig. 8 Replace Operation

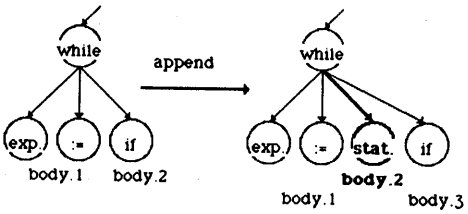

Fig. 9 Append Operation

３．５　ＬＤＥ内メッセージ

オブジェクトはメッセージによって他のオブジェクトと通信する．メッセージにはＬＤＥ内のオブジェクトへのメッセージと他のＬＤＥへのメッセージの２種類がある．本節では前者について述べる．

メッセージは以下の形式を取る．

```
message     ::= destination method-name parameters
destination ::= link | tool | context
                | self | super
link        ::= link-type-name . link-number
tool        ::= tool tool-name
context     ::= context
self        ::= self
super       ::= super . super-class-name
```

（下線は、Ｄ言語における予約語を表す．）

destination はメッセージが送られるオブジェクトを示す．

link はリンクによって結合されているオブジェクトにメッセージを送ることを表す．リンクを通じてメッセージを送る場合、メソッド名はリンククラスに定義されていなければならない．継承とリンククラスの制約により、メッセージを送られたオブジェクトがメッセージを常に受け取ることが保証される．

context と tool は、それぞれコンテキスト・オブジェクト、ツール・オブジェクトへメッセージを送ることを表す．

self は自分自身の、super は上位クラスに定義されているメソッドを起動することを表す．

３．６　宣言の取り扱い

ＬＤＥでは構文・意味チェックの実現の容易化と編集操作の統一化のため、プログラム中で宣言された変数や手続きを構文要素と同様なオブジェクトと考える方法をとっている．

多くの言語指向エディタや統合化プログラミング環境では、

```
function P ( X : in REAL ; Y : in out REAL )
                                    return INTEGER ;
```

のように宣言された関数ｐの参照は、関数呼出しを表すノードの名前属性にｐを入力する方法がとられる．

これに対しＬＤＥでは、上記のような宣言から関数呼び出しクラスのサブクラスであるクラスｐを定義する．このクラスは構文木のブロックや手続きなどのノードの属性である記号表に登録される．関数ｐの参照はクラスｐのノードへの置き換えによって実現される．図１０にクラスｐの置き換えによってできるテンプレートを示す．仮パラメタＸに対応してＸというリンク型名のリンクが生成される．仮パラメタＸのモードは in なのでノードクラスは expression となる．同様にＹのモードは in out なのでノードクラスは variable となる．

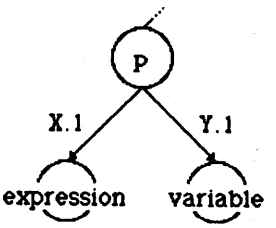

Fig. 10 Template of "p" Class

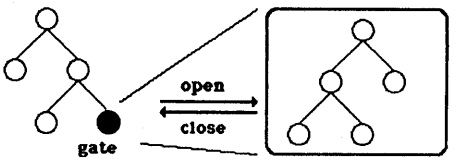

Fig. 11 Example of Gate

４．　ＬＤＥ間の処理

４．１　構文木間の関係

あるノードが別の構文木全体を表しているとき、そのノードをゲート[10]という．例えば、図１１のようにディレクトリを表している構文木の葉ノードは、ひとつの構文木で表現されるあるプログラム単位を表している．

ＭＵＬＳは構文木間の関係をゲートを用いて表す．またゲートはＬＤＥの保持する構文木の切り換え点となる．図１１において、ディレクトリ用のＬＤＥのひとつのゲートにカーソルを動かしｏｐｅｎというコマンドを入力することにより、ＬＤＥの保持する構文木はプログラム用データベースへ書き込まれ、そのゲートが表すプログラム単位の構文木およびＬ情報・Ｔ情報が取り込まれる．逆に、切り換わったＬＤＥにおいてｃｌｏｓｅコマンドを入力したとき、上記の逆の処理により、ＬＤＥはもとの状態に戻る．

４．２　ＬＤＥ外メッセージ

ある構文木が別の構文木に及ぼす処理は、ＬＤＥ外へのメッセージによって実現される．ＬＤＥ外メッセージには、

(1) ゲートからそれが表す構文木用のＬＤＥのコンテキスト・オブジェクトへのメッセージ、と

(2) ＬＤＥからその構文木を表しているゲートへのメッセージ、

の２種類がある．ＬＤＥ外メッセージはＬＤＥの切り換えを必要とするため、処理時間がかかりユーザのコマンドの入力に対する反応が悪くなるといった問題があるので、即時には処理されず、メッセージを送られた側のＬＤＥがｏｐｅｎされたとき初めて処理される．

５．　Ｄ言語

１つのＬＤＥを実現するためには、Ｄ言語を用いて１つのＬ記述といくつかのＴ記述を作成しなければならない．Ｔ記述は既にあるものを用いてもよい．Ｌ記述、Ｔ記述ともに、宣言部とクラスを定義する部分からなる．

５．１　Ｌ記述

宣言部と言語の構文単位に対するクラスの定義からなる．宣言部は、以下の情報からなる．クラスの定義については３．３で説明したのでここでは省略する．

(1) 名前：言語の名前．

(2) インポート：この言語がインポートする言語の名前．あるＬ記述において定義されたクラスの名前の有効範囲は、そのＬ記述内に限られるので、別のＬ記述内で定義されたクラスを参照する場合、あらかじめそのＬ記述の言語名を宣言しておかなければならない．なお、記号表やゲートといった言語間に共通な機能はクラスとして定義されており、インポートして上位クラスとすることにより、これらの機能を容易に利用できる．

(3) ツール：この言語で使用するツールの名前．

(4) ルート：プログラムの新規作成時に生成されるノード．
(5) コマンド：この言語のＬＤＥで使用可能なコマンド．コンテキスト・オブジェクトのメソッドとして定義され、言語用データベースに保持される．この言語に対するＬＤＥの切り換え時にコンテキスト・オブジェクトが持つメソッドに追加される．

言語の宣言部の例を図１２に示す．

５．２　Ｔ記述

宣言部とツールに対するクラスの定義からなる．宣言部は、以下の情報からなる．
(1) 名前：ツールの名前．
(2) コマンド：このツールを使用するＬＤＥで使用可能なコマンド．コンテキスト・オブジェクトのメソッドとして定義される．

６．　まとめ

本稿では、環境内の言語・ツールに関する情報を取り換えることにより複数の言語の各々に対し専用化されたプログラミング機能を提供する拡張型統合化プログラミング環境ＭＵＬＳについて述べた．また、ＭＵＬＳ内の処理に導入したオブジェクト指向プログラミングの考え方について述べた．すなわち、ツールおよびプログラムの内部表現である属性付き抽象構文木の各ノードをオブジェクトとみなし、オブジェクト間のメッセージの授受によって、ユーザの操作に対する処理を行なう．

複数の言語を扱う機能とオブジェクト指向の導入により、
(1) 複数の言語を用いたプログラムの作成の支援、
(2) 木構造のデータを扱う機能に対する一貫したユーザインタフェースの提供と実現の容易化、
(3) 環境の機能の拡張・変更の容易化、
といった利点が得られる．

ＭＵＬＳのプロトタイプ・システムをUNIX上のFrantz-lispで実現した．Ａｄａ、Ａｄａ構成管理、記述言語自身、をＬ記述で実現した．ツールとしてプリティ・プリンタ、Ｌ記述からクラスを生成するトランスレータ、コード生成ツール、デバッガを実現した．

今後の課題としては、種々の言語・ツールの実現と評価、ユーザインタフェースの改良などがあげられる．

```
language    Ada
        import      symbol-table
        tool        pretty-printer,
                    code-generator
        root        comp-unit
        command
                open-subunit ()
                        (...)
                ...
class definition
        ...
end
```

Fig. 12 Ada Language Description

謝辞　本研究を進めるにあたり、有益な御意見を頂いた広島大学工学部第二類平川正人博士、システムの実現に際して御協力頂いた情報システム研究室の佐藤康臣氏、永良裕氏、ならびに日頃御討論頂く同研究室の諸氏に感謝する．

文　献

[1] W. Teitelman and L. Masinter: "The Interlisp Programming Environment", Computer, vol.14, no.4, pp.25-34 (1981).

[2] T. Teitelbaum and T. Reps: "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment", Comm. ACM, vol.24, no.9, pp.563-573 (1981).

[3] T. A. Standish and R. N. Taylor: "Arcturus: a Prototype Advanced Ada Programming Environment", Proc., ACM SIGSOFT/SIGPLAN Symp. on Practical Software Development Environments, pp. 57-64 (Apr. 1984).

[4] N. Delisle, D. Menicosy: "Viewing a Programming Environment as a Single Tool", Proc., ACM SIGSOFT/SIGPLAN Symp. on Practical Software Development Environments, pp. 49-56 (Apr. 1984).

[5] T. Reps and T. Teitelbaum: "The Synthesizer Generator", Proc., ACM SIGSOFT/SIGPLAN Symp. on Practical Software Development Environments, pp. 42-48 (Apr. 1984).

[6] S. P. Reiss: "An Approach to Incremental Compilation", Proc., ACM SIGPLAN Symp. on Compiler Construction, pp. 144-156 (June 1894).

[7] "ALOE Users' and Implementors' Guide", Dept. Computer Science, Carnegie-Mellon Univ. (Oct. 1984).

[8] T. Tenma, H. Tsubatani, M. Tanaka, and T. Ichikawa: "A Genetration System for Language-Oriented Editors", Proc., COMPSAC'86, pp.105-112 (Oct. 1986).

[9] A. Goldberg and D. Robson: Smalltalk-80:The Language and Its Inplementation", Addison-Wesley (1983).

[10] V. Donzeau-Gouge, G. Kahn, B. Lang, and B. Melese: "Document structure and modularity in Mentor", Proc., ACM SIGSOFT/SIGPLAN Symp. on Practical Software Development Environments, pp. 141-148 (Apr. 1984).