Adaとその記号処理への応用

寺島 元章　　杉山裕一　　山地正記
〔電気通信大学計算機科学科〕　〔朝日新聞社〕　〔日本電気〕

1. はじめに

P-code インタプリタ方式による Ada 処理系*の概要とこの処理系の使用経験から得た Ada の特徴、Ada プログラミングの意義とその応用分野、特に記号処理への応用について述べる。この処理系は、Pascal 8000 [9] で記述された移設可能な処理系で、仮想スタック計算機の中間言語である P-code を生成するコンパイラと P-code を解釈実行するインタプリタとから成る。この処理系が Ada のいくつかの機能を制限したものであること、また、実行速度が遅いインタプリタ方式であることにもかかわらず、この処理系を使用して Ada プログラムの作成が行なわれており、これまでに、名前表の管理、リスト処理等を行う応用プログラムが作成されている。

2. 言語 Adaについて

Ada [1,2] は、米国国防総省（DoD）の組込み型計算機システム用共通高水準言語の制定過程での提案「Steelman」の要求仕様を満すために設計された言語である。Ada はその設計目標として、プログラムの信頼性と保守性に重点が置かれているのでプログラムの書き易さより読み易さが強調されており、またプログラマーへの配慮から言語の規模を比較的小さくまとめてあり、既成の計算機の機能特徴を考慮して最適な目的コードが生成できる様にインプリメンテーション上の効率についても追求されている。

Ada は、強い意味で型（type）付けされた言語である。変数、定数等の客体（object）は、それぞれ型が（翻訳時に）決まっていなければならないし、名前や式を評価して得られる値には型があるなど Pascal [7] の型との類似点を多く持つ。Ada では、副型（subtype）と言われるものが導入され、型は、演算を識別するための基本型（base type）と取り得る値の識別のための副型に機能的に分類される。前に定義された型と構文上同じ定義を行なって生成した型は、前述の型と別の型になる。この問題を解決するために導出型（derived type）と言うものが導入され、その型を導出した親型（parent type）の性格をその導出型が受け継ぐことができ、また親型の手続きなどを共用できる様になった。

型は、スカラー型（列記型，整数型，実数型），合成型（配列型，レコード型），参照型（access type），私用型（private type）とタスク型に分類される。Pascal と比較すると、集合（set）がなくなり、かわりに、並行処理を記述できシステム記述用機能をもつタスクと、包（package）中で使用しユーザーからは無関係な内部（データ）構造をユーザーから保護するというデータ抽象化支援機能をもつ私用型が加わった。また、各型の性格を調べる属性（attribute）や合成

---

*この処理系の作成は 1981 年 1 月から開始したために Preliminary Ada [2] の仕様に基づいているが、機能の追加等の作業は本年度も継続して行なっており、除々に新版（1981年7月版）[1] に仕様を変更しつつある。また、本稿で述べる事項はすべて新版の仕様で記述してある。

分値で作られた合成体の値を表わす aggregate、参照値（access value）を返す配置子（allocator）などがある。

Adaプログラムは1個又は複数個のプログラム単位から成り、それらは分割翻訳（separate compilation）可能である。プログラム単位は、副プログラムである、型定義、関数定義等の論理的に関連づけられた実体（entity）の集体である包、又はタスクのいずれかである。

制御構造は、loop, and then, else or exception（例外）等の導入により Pascal のそれより豊富になった。また式の評価方法についてその細部の規定が行なわれた。overloading（サブプログラム名，変数名、列記型の定数名などの名前を多重定義することであり、型の識別ができれば多重定義された名前であってもその名前を一義的に識別できることから、一つの名前でいくつものを代表させる機能）、例外（桁あふれ，参照誤り等のエラーから優雅にかつ統一的に退出させるための機能）、汎用（generic、型を汎用化させる機能で、型の相異を無視して処理を行う必要がある場合に使用する）、この他に library，表現明細（representation specification）等の機能があるが、本稿に直接関連する事項でないので省略する。

3. UEC_ADA

3.1 概要

この処理系は、Preliminary Ada [2] の仕様に基づき、そのいくつかの機能を制限した処理系（大学名をとってUEC－Adaと称する）で、当初、電通大情報処理センターのM170上に作成された。UEC_Ada は Pascal 8000 で記述されており、P-code を生成するコンパイラ部が約2000行、P-code を解釈実行するインタプリタ部が約300行である。このコンパイラ部とインタプリタ部は分割可能となっている。コンパイラが生成する P-code は、基本 P-code [8] を拡張したものであるが基本的には純スタック演算に基づいている（付録2参照）。UEC_Adaの作成にあたり、Pascal の P-code コンパイラを参考にした主な理由は、

(1) Ada の翻訳が Pascal と同じ1パスで行なえること
(2) 可視制限機能*（visibility restriction）を除けば Ada の意味解析部は Pascal の解析部と同様の手続きとなる。
(3) UEC_Ada に当面必要な機能は Pascal（少なくとも Pascal S）の機能を包括すれば十分である。
(4) 比較的短期間に処理系が作成できる。
(5) 移設可能（portable）であること。

による。

UEC_Ada の現時点での機能を付録1に構文記法で示す。束縛（constraint）実数，aggregate，汎用体，タスク（並行処理）、入出力の大部分の機能、分割翻訳，library と表現明細などが組み込まれていない。

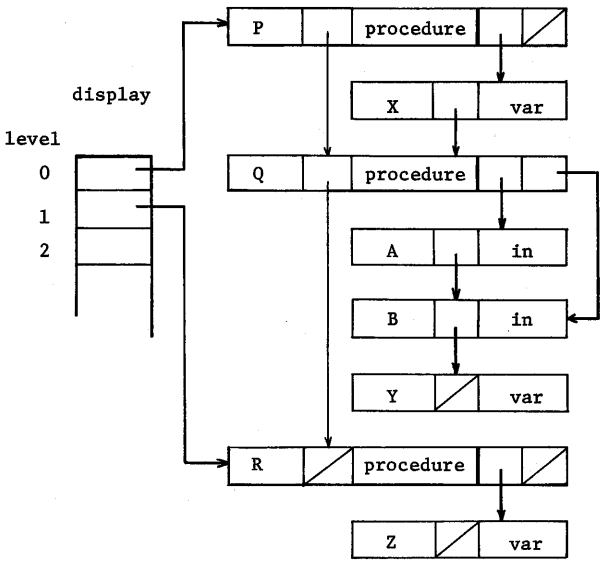
次に、UEC_Ada 処理系作成上で問題になった点とその対処法について述べる。

---

*スコープの外側にある名前を内側のらは見えなくする機能のことで新版 [1] の仕様のらは除かれている。

3.2 名前表

名前表は、宣言において名前（識別子）に付けられた属性を、後に名前を鍵にしてその名前の属性を探索するのに用いられる。可視制限機能がない場合には名前表の管理はきわめて簡単なものになる。名前表を図1に示す。この名前表を利用して overload された名前を探索する場合、display をたどる操作が追加され、display の水準が深くなればそれに比例して探索効率が低下するので、overload された名前を効率良く探索するためには、display の水準（use 節により可視となった名前も同時に探索可能にするためにこの様な名前に対して特別の水準を与える）と名前との対を鍵とする表探索が必要である。2個の要素の組に対して属性が付けられ、この組を鍵としてその属性を探索できる連想計算機能（associative computing）をもつ処理系では、上述の探索は効率の良いものとなろう。




```
procedure P is
X: integer;
   procedure Q (A , B: integer) is
   Y: integer;
   begin
     --
   end Q;
   procedure R is
   Z: integer;
   begin
     --
   end R;
begin
  --
end P;
```

図1　名前表の構造（変数名，仮引数名の属性に関する表の部分を除く）

3.3 制御構造と P-code

現在使用中の P-code は、純スタック演算が基本である。このため、loop 中の exit 文の処理，例外処理，goto 文の処理ではスタックポインタの補正が必要である。この補正のための命令と、例外処理のための複合命令を P-code に追加した（付録2参照）。

3.4 例外処理

例外が生じた場合に引用される例外処理部（exception handler）は動的に決定される。これを実現するために、例外処理部をもつ副プログラムを実行する場合に例外処理部の番地をスタックに先づ登録する方式を採用した（図2参照）。また、例外の伝達（propagate，例外がその規定された範囲を越えて伝達されること）を可能にするため、プログラムの例外名に対して処理系が一義的な名前に付けなおすことを行なっている。

4. Adaプログラミング

図4（10P）は、当学科の3年次のPASCAL演習の課題の一つである「二分探索木の作成とその探索」をAdaでプログラミングしたものである。プログラミングツールとしてAda，PASCAL等の強い意味で型付けされた言語を選ぶことの意義はそのプログラムの信頼性にある。これらの言語では、すべての客体に型が付けられており，それらは実行前に宣言されることから，変数、定数等の客体に対する演算や代入操作は翻訳時にその構文検査が行なわれ不法な演算等が検出される。また，配列の添字範囲外の参照、又は参照誤り等の実行時の型侵害の検査* が行なわれ，プログラムの信頼性を向上につながっている。

次に同じ型付け言語であるPASCALとAdaとそのプログラミングにより比較すると

(1) PASCALと比較してプログラムが短かく記述できる。

- PASCALにないAda固有の制御構造である if と end if，elsif，and then，or else を使用することにより不要な begin end を追放でき、又 if 文の入れ子の数を減じることが可能になったこと。
- case 文で式を評価した値が選択子（choice）のいずれとも合致しない場合の例外（others）が使用できること。
- loop の作業変数の宣言が不要であること。
- aggregate の使用や宣言において代入文を用いることで合成体への値の代入が容易に記述できること。
- 例外（exception）の使用により例外処理の記述（10Pの例題ではファイルの終端の検出（eof））が平易にかつ統一的に行なえること。

などの理由によるものと考えられる。

(2) プログラムが読み易くなったこと。

これは(1)の副次的な結果であるとも言えるが、

- 予約語 is，return，exit 等の導入に見られる様にプログラム構文が日常語に近くなったこと。
- 配列要素、副プログラム引用、型指定式（qualified expression）に現われるかっこはすべて（）で表記される。
- 識別子（discriminant）の束縛，aggregate，case 文，例外中の「…は…」又は「…ならば…」は、簡明な複合記号 => で表記される。

などが理由として考えられるが，Adaがその言語の設計目標に「書き易さより読み易さ」を追求したことの具体的現われであろう。

5. Adaの記号処理への応用

UEC_Ada処理系を用いて今までに作成した応用プログラムは、ハッシュ法による名前表の作成とその探索、クロスリファレンスの出力という様な、文字処理用の比較的小さいソフトウェアツールとしてのものが多い。これは，現Ada処理系がP-codeインタプリタ方式に基づき、かつAdaのいくつかの特徴を制

---

*この検査は実行時のオーバーヘッドとなることから，この実行時検査を省略する（翻訳）オプションが用意されている。PASCALでは翻訳オプションで，Adaでは，pragma SUPPRESS により指定できる。

限したことの処理系自体の問題によるものであろうが、行数にして300行を越える様な応用プログラムは作られていない。

Adaは、参照型と呼ばれるデータ型をもっており、これは、リスト構造を(番地を直接にリンクに用いて)表現できる。したがって、単にリスト処理だけを目的とする様な応用プログラムは、Adaで十分記述可能であり、リスト処理を含む実用的な記号処理プログラム(Lispは処理系)をAdaで記述する場合の問題点も次の理由(機能)により特に見当たらない。

(1) 副プログラムは、その分割翻訳が可能でありかつreentrantであること。
(2) 可視規則により大域変数名が見える(扱える)こと。
(3) (1)(2)の事由から副プログラムの引用手順(calling sequence)がわかれば、他の言語(例えばアセンブラ)との結合が可能であること。
(4) 例外処理、割込み処理、並列処理の機能が使えること。
(5) 参照型各体に対する記憶配置(storage allocation)管理が行なわれる。

5. まとめ

UEC-Adaは、現在、当学科のMELCOM-COSMO 700 III上で稼動しており、翻訳速度は10~15行/秒(図5参照)である。この処理系を実用に供するためには、P-codeの改良、インタプリタ部の機械語化、さらには直接機械語を生成するコンパイラ式への移行が必要であろう。

```
PROCEDURE LISP_SYSTEM IS
TYPE S_STRING(CLNGTH:INTEGER);
TYPE L_STRING IS ACCESS S_STRING;
TYPE S_STRING (CLNGTH:INTEGER := 4) IS
    RECORD ITEM: ARRAY(1..CLNGTH) OF CHARACTER;
           LINK: L_STRING
    END RECORD;
TYPE CELL (CELL_TYPE:BOOLEAN);
TYPE LINK IS ACCESS CELL;
TYPE CELL (CELL_TYPE) IS
    RECORD CASE CELL_TYPE IS
        WHEN ID => VAL: LINK;
                   PNAME: L_STRING;
             SINT => VAL: INTEGER;
             SREAL => VAL: REAL;
             PAIR => CAR,CDR: LINK;
             SUBR,FSUBR => VAL: INTEGER;
                           PNAME: L_STRING;
             EXPR,FEXPR => VAL: LINK;
                           PNAME: L_STRING;
    END CASE;
END RECORD;
```

図3. Lispデータの宣言(Lispインタプリタでの使用例)

References

Adaについて

1. Reference Manual for the Ada Programming Language. United States Department of Defence (U.S. DoD), (July, 1980).
プログラミング言語 Ada 基準文法書(1980年12月) 共立出版(Reprint)
2. Preliminary Ada Reference Manual. U.S. DoD. SIGPLAN Notices, 14, 6 (June, 1979).
3. Barnes, J. G. P. An Overview of Ada. SOFTWARE 10, 11 (Nov., 1980).
4. Proceedings of the ACM-SIGPLAN Symposium on the Ada Programming Language. SIGPLAN Notices, 15, 11 (Dec., 1980).
5. 筧捷彦 Ada-米国国防総省新言語, 情報処理 21, 9 (1980年9月) 975-979
6. プログラミング言語-PASCALとAda, 情報処理 22, 2 (1981年2月)
この小特集の中にAdaとその処理系に関する記述とがある。

Pascalとp-codeについて

7. Wirth, N. The programming language PASCAL. Acta Informatica, 1, 1 (1971) 35-63.
8. Nori, K. V. et. al. The PASCAL <P> Compiler; Implementation Notes, Berichte Nr. 10, Institut für Informatik, Eidgenossische Technische Hochschule, Zurich (1976).
9. Hikita, T. and Ishihata, K. PASCAL 8000 Reference Manual. Technical Report 76-02, Dept. of Information Science. University of Tokyo (1976).

UEC-Adaについて

10. 杉山裕一 Adaの中間言語への翻訳 電気通信大学計算機科学科卒業論文(1981年1月)
11. 山地正記 Ada処理系の試作 電気通信大学計算機科学科卒業論文(1981年1月)

APPENDIX 1. Syntax Summary of UEC-ADA

1. Syntax Notation

Syntax notation of UEC-ADA is described using a simple variant of BNF. In particular,

(a) Lower case words denote syntactic categories.
(b) Underscored words denote reserved words.
(c) Square brackets [] enclose optional items.
(d) Braces enclose {} a repeated item including zero time.
(e) Vertical bars | separete alternative items.

Note: Each number used in heading corresponds to the section of ADA reference manual [1].

2. Lexical Elements

2.1 Character Set
upper case letters
digits
special characters
" # ' ( ) * + , - . / : ; < > |
the space haracter

2.2 Delimiter of compound symbols
=> ** := /= >= <= << >> =: :=

2.3 identifier ::=
letter {[underscore] letter_or_digit}

letter_or_digit ::= letter | digit

letter := upper case letter

2.4 numeric_literal ::=
decimal_number | based_number

decimal_number ::=
integer [.integer][exponent]

integer ::= digit {[underscore] digit}

exponent ::= E [+] integer | E - integer

based_number ::=
base # based_integer

based_integer ::=
extended_digit {[underscore] extended_digit}

base ::= integer

extended_digit ::= digit | letter

2.6 character_string ::= " character "

3. Declarations

3.1 declaration ::=
object_declaration
| type_declaration
| subprogram_declaration
| exception_declaration ::=

object_declaration ::=
idetifier_list : [constant]
subtype_indication [:=expression];
| identifier_list : [constant]
array_type_difinition [:= expression];

identifier_list ::=
identifier {,identifier}

3.3 type_declaration ::=
type identifier is type_definition;

type_definition ::=
array_type_definition
| record_type_definition
| access_type_definition
| derived_type_definition

3.4 derived_type_definition ::=
new subtype_indication

subtype_indication ::=type_mark

type_mark ::= type_name

3.6 arrar_type_definition ::=
array index_constraint of type_mark

index_constraint ::=
(discrete_range {,discrete_range)}

discrete range ::= range

range ::=
simple_expression .. simple_expression

3.7 record_type_definition ::=
record
component_list
end record

component_list ::= component_declaration

component_declaration ::=
identifier list :
subtype_indication [:=expression]

3.8 access_type_definition ::=
access sub_type_indication

4. Names and Expressions

4.1 name ::=
identifier | indexed component
| selected_component | attribute

indexed_component ::=
name(expression {,expression)}

attribute ::= name'identifier

selected_component ::=
name.identifier | name.all

4.2 literal ::=
numeric_literal | character_string | null

4.4 expression ::=
relation {and relation}
| relation {or relation }
| relation {xor relation}
| relation {and then relation}
| relation {or else relation}

relation ::=
simple_expression
[relational_operator
simple_expression]
| simple_expression [not] in range

simple_expression ::=
[unary_operator] term
{adding_operator term}

term ::= factor
{multiplying_operator factor}

```
        factor ::= primary [**primary]

        primary ::=
        literal | name | allocator
      | function_call | qualified_expression
      | (expression)
```

4.5 `logical_operator ::= and | or | xor`

```
relational_operator ::=
   = | /= | < | <= | > | >=
```

```
adding_operator ::= + | -
```

```
unary_operator ::= + | - | not
```

```
multiplying_operator ::= * | / | mod
```

```
exponentiating_operator ::= **
```

4.7 `qulified_expression ::=`

```
   type_mark(expression)
```

4.8 `allocator ::= new type_mark`

5. Statements

5.1 `sequence_of_statements ::=`

```
   statement {statement}
```

```
statement ::=
    {label} simple_statement
   |{label} compound_statement
```

```
simple_statement ::=

     null;
   | assign_statement | exit_statement
   | return_statement | procedure_call
   | raise_statement
```

```
compound_statement ::=
   if statement | case statement
   loop statement
```

```
lebel ::= <<identifier>>
```

5.2 `assign_statement ::=`

```
   variable_name := expression
```

5.3 `if_statement ::=`

```
   if condition then sequence_of_statements
     {elseif conditon then
                  sequence_of_statements}
     [else sequence_of_statement]
   end if
```

```
condition ::= Boolean_expression
```

5.4 `case_statement ::=`

```
   case expression
     {when choice {|choice} = >
           sequence_of_statement}
   end case;
```

```
choice ::=
     simple_expression | discrete_range
   | others
```

5.5 `loop_statement ::=`

```
   [iteration clause]
   basic_loop [loop_identifier];
```

```
basic_loop ::=
   loop
      sequence_of_statements
   end loop
```

```
iteration_clause ::=
   for loop_parameter in [reverse]
   discrete_range | while condition
```

```
loop_parameter ::= identifier
```

5.7 `exit_statement ::=`

```
   exit [loop_identifier] [when condition];
```

5.8 `return_statement ::=`

```
   return [expression] ;
```

6. Subprograms

6.1 `subprogram_declaration ::=`

```
   subprogram_specification
```

```
subprogram_specificaton ::=

     procedure identifier [formal_part]
   | function designator [formal_part]
     return subtype_indication
```

```
designator ::= identifier
```

```
formal_part ::=
   (parameter_declaration
      {;parameter_declaration})
```

```
parameter_declaration ::=
   identifier_list : mode subtype_indication
```

```
mode ::= [in] | out | in out
```

6.3 `subprogram_body ::=`

```
   subprogram_specification is
      declative_part

   begin
      sequence_of_statements
   [exception {exception_handler}]
   end [designator];
```

6.4 `procedure_call ::=`

```
   procedure_name [actual_parameter_part];
```

```
function_call ::=
   function_name actual_parameter_part
```

```
actual_parameter_part ::=
   (parameter_association
       {,parameter_association})
```

```
parameter_association ::= actual_parameter
```

```
actual_parameter ::= expression
```

11. Exceptions

11.1 `exception_declaration ::=`

```
   identifier_list : exception ;
```

11.2 `exception_handler ::=`

```
   when exception_choice {|exception_choice}
         => sequence_of_statements
```

```
exception_choice ::= exception_name | others
```

11.3 `raise_statement ::=`

```
   raise [exception_name]
```

APPENDIX 2. Extended P-code Summary

Symbols used in the following symbolic description mean:

S : stack T : stack pointer
D : display stack PC : program counter

PUSHADR	X	Y	push stack position, symbolically:	T ← T+1; S(T) ← D(X)+Y
PUSHVAL	X	Y	push value;	T ← T+1; S(T) ← S(D(X)+Y)
PUSHIND	X	Y	push value indirectly;	T ← T+1; S(T) ← S(S(D(X)+Y))
UPDATE	X	Y	update display	
CALLSF		Y	call standard function, namely ABS, SQR, ROUND, TRUNC, SIN, COS, LN, SQRT, EOLN and EOF.	
OFFSET		Y	increment (decrement) S(T) by Y;	S(T) ← S(T)+Y
JUMP		Y	jump to Y;	PC ← Y
JUMPC	X	Y	conditonal jump;	(if S(T)=Y then T ← T-1; PC ← Y fi)
CASE			code for case	
INTG		Y	code to indicate integer Y.	
LINKD		Y	code to indicate link address Y.	
TEST			test top two contents greater	(if S(T-1) > S(T) then EXITPROC)
TESTGEI			test top two contents greater or equal and increment	(if S(T-1) ≥ S(T) then EXITPROC else S(T-1) ← S(T-1)+1 fi)
TESTL			test top two contents low	(if S(T-1) < S(T) then EXITPROC)fi)
TESTLED			test top two contents low or equal and decrement Note: These four codes are generated for in loop compilation.	(if S(T-1) ≤ S(T) then EXITPROC else S(T-1) ← S(T-1)-1 fi)
MARK		Y	mark stack for frame reservation;	T ← T+5; S(T) ← Y
INDEX		Y	code for index;	S(T-1) ← S(T-1) + (S(T)-Y')*Y Y" is next located code INTG's Y.
CALL	X	Y	call procedure X is level for display and Y is frame size.	
PUSHBLK		Y	push block content(s) Y is block size.	
COPYBLK		Y	copy block content(s) Y is block size.	
PUSHINT		Y	push integer Y immediately;	T ← T+1; S(T) ← Y
PUSHREAL		Y	push real Y immediately;	T ← T+1; S(T) ← Y
CVTREAL			convert stack top content into real value	
CVTINT			convert stack top content (real number assumed) into integer	
READ		Y	call input function to read integer, real and character data	
WRITESTR			write string	
WRITEI			call output function to write integer, real and character data	
STOP			terminate execution	
EXITPROC			exit from procedure	
EXITF			exit from function	

LOAD		load stack top content itself;	S(T) ← S(S(T))
EXIT		exit from procedure (or function) without PC reset	T ← B-1; B ← S(B+3) B is stack base pointer of the procedure.
IN		code of in;	S(T-2) ← S(T-2)≥(S(T-1)) (S(T-2)≤S(T)); T ← T-2
RAISE		code of raise.	
ALLOC		allocate block;	BT ← BT-Y; T ← T+1; S(T) ← BT+1 BT is stack pointer for block.
CMPB	Y	compare block Y is block size.	

arithmetic and logical operations

NOT	S(T) ← ¬S(T)	
MINUSREAL MINUSINT	S(T) ← -S(T)	
EQREAL EQINT	S(T-1) ← S(T-1)=S(T);	T ← T-1
NEREAL NEINT	S(T-1) ← S(T-1)≠S(T);	T ← T-1
NGEREAL NGEINT	S(T-1) ← S(T-1)<S(T);	T ← T-1
NGREAL NGINT	S(T-1) ← S(T-1)≤S(T);	T ← T-1
NLEREAL NLEINT	S(T-1) ← S(T-1)>S(T);	T ← T-1
NLREAL NLINT	S(T-1) ← S(T-1)≥S(T);	T ← T-1
ADDREAL ADDINT	S(T-1) ← S(T-1) + S(T);	T ← T-1
SUBREAL SUBINT	S(T-1) ← S(T-1) - S(T);	T ← T-1
MULREAL MULINT	S(T-1) ← S(T-1)*S(T);	T ← T-1
DIVINT DIVREAL	S(T-1) ← S(T-1) / S(T);	T ← T-1
MOD	S(T-1) ← S(T-1) mod S(T);	T ← T-1
AND	S(T-1) ← S(T-1)∧S(T);	T ← T-1
OR	S(T-1) ← S(T-1)∨S(T);	T ← T-1
XOR	S(T-1) ← S(T-1) xor S(T);	T ← T-1
POWERR POWERI	S(T-1) ← S(T-1)**S(T);	T ← T-1

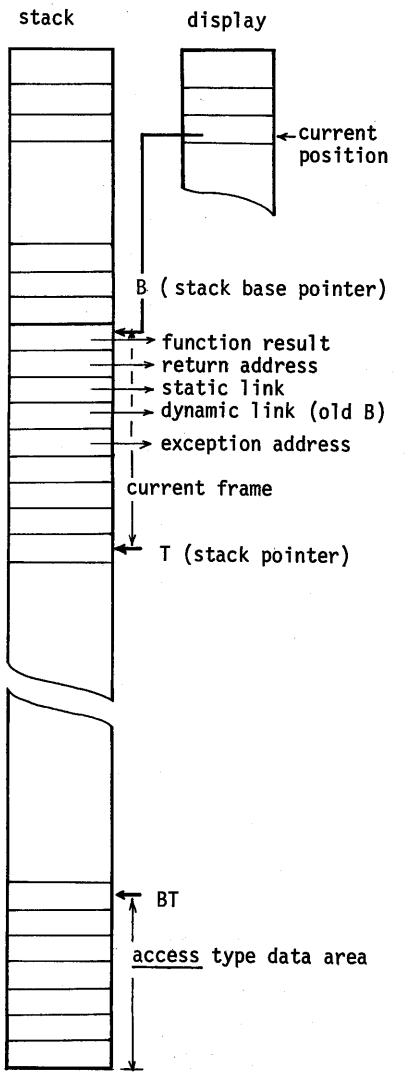


Fig. 3. Stack utilization of UEC_Ada.

```
PROCEDURE BINARYTREESEARCH IS
  TYPE ALFA IS ARRAY (1..10) OF CHARACTER;
  TYPE REF IS ACCESS
       RECORD KEY : ALFA;
              INFO: INTEGER := 1;
              LEFT,RIGHT: REF := NULL;
       END RECORD;
  NAME,CLEAR : ALFA;
  F,EON : BOOLEAN;
  P,G,V : REF;
  C,NO : INTEGER := 0;  I : INTEGER;
  CH    : CHARACTER;
  END_OF_NAME:EXCEPTION;
  PROCEDURE GETNAME IS
    I:INTEGER;
  BEGIN WHILE CH<"A"  OR CH>"Z"
        LOOP GET(CH);
        END LOOP;
        I := 0;
        NAME := CLEAR;
        WHILE CH>="A" AND CH<="9"
              LOOP
                IF I<ALFA'LAST THEN I:=I+1;
                                    NAME(I):=CH;
                END IF;
                GET(CH);
              END LOOP;
  EXCEPTION
  WHEN END_OF_FILE =>
    RAISE END_OF_NAME;
  END GETNAME;
  FUNCTION GREATER(N1,N2:ALFA) RETURN BOOLEAN IS
  BEGIN
    FOR I IN 1..ALFA'LAST
      LOOP IF N1(I)>N2(I) THEN RETURN TRUE;
           ELSIF N1(I)<N2(I) THEN RETURN FALSE;
           END IF;
      END LOOP;
    RETURN FALSE;
  END GREATER;
  PROCEDURE INORDER(R:REF) IS
    I:INTEGER;
  BEGIN IF R/=NULL
          THEN INORDER(R.LEFT);
               NO:=NO+1; PUT(NO); PUT(".   ");
               FOR I IN 1..ALFA'LAST LOOP PUT(R.KEY(I)); END LOOP;
               PUT(" "); PUT("-----");
               PUT(R.INFO); PUT(" TIME(S)"); PUT(NEWLINE);
               INORDER(R.RIGHT);
       END IF;
  END INORDER;
  PROCEDURE SEARCH_AND_INSERT(Q: IN REF; R: IN OUT REF) IS
  BEGIN IF R=NULL
          THEN R:=NEW REF;
               R.ALL := Q.ALL;
               F:=FALSE;
          ELSIF GREATER(R.KEY,Q.KEY)
               THEN C:=C+1;
                    SEARCH_AND_INSERT(Q,R.LEFT);
          ELSIF GREATER(Q.KEY,R.KEY)
               THEN C:=C+2;
                    SEARCH_AND_INSERT(Q,R.RIGHT);
          ELSE C:=C+2; F:=TRUE; G:=R;
        END IF;
  END SEARCH_AND_INSERT;

BEGIN      -- MAIN PROGRAM
  V:=NULL;
  FOR I IN 1..ALFA'LAST LOOP CLEAR(I):=" "; END LOOP;
  P := NEW REF;
  CH := " ";
  GETNAME;
  BEGIN
       LOOP P.KEY := NAME;
            SEARCH_AND_INSERT(P,V);
            IF F THEN G.INFO := G.INFO+1; END IF;
            GETNAME;
       END LOOP;
  EXCEPTION
    WHEN END_OF_NAME => NULL;
  END;
  INORDER(V);
  PUT(NEWLINE);PUT("*** TOTAL COMPARATIVE NUMBER OF ITEM :");
  PUT(C); PUT(NEWLINE);
END BINARYTREESEARCH;
;;
```

図4　二分探索木の Adaプログラム

プログラム	翻訳時間 (秒)	実行時間 (秒)
ハノイの塔 (20行)	1.5	0.4 (n=5)
パスカルの三角形[6] (16行)	1.7	7.8
二分探索木[図4] (83行)	4.2	48.0*
ハッシュによる名前表 (110行)	6.5	18.2*
クロスリファレンス出力 (140行)	10.7	—
Lisp インタプリタ (250行)	21.1	—

*データ入力は二分探索木のプログラムを用いた。

図5　UEC_Ada処理系の処理速度