UDEC-II における直接実行アルゴリズムの設計†

板野肯三†† 佐藤 豊†††

ソースプログラムの直接実行型計算機 UDEC-II のために，PASCAL サブセット用の直接実行アルゴリズムを，通常の構文解析を行うための生成規則に意味の解釈や実行の指定を組み合わせて記述する実行文法を用いて開発した．特に，使用される意味手続きを機能的な観点と使用頻度による最適化の観点からプリミティブとして再設計し，アルゴリズムの記述性の改良を行った．さらに，意味処理部のうちから先行制御の可能ないくつかの機能を分離して，独立のモジュールとしてパイプライン構造の各部に分散し，実行の高速化を計ると同時に，これを直接実行アルゴリズムのより形式的な記述に反映させた．

1. はじめに

高水準のプログラミング環境を実現する基礎として，実用的な性能でソースプログラムを直接実行することができる，直接実行型計算機 UDEC (Universal Direct-Execution Computer) を試作した[1)~8)]．UDEC は，パイプライン型に結合された字句解析部，構文意味認識部，意味解析実行部を中心として構成されている．特定の言語に対する直接実行型計算機を実現するには，これらのモジュールの制御テーブルや制御記憶に，対象言語の構文や意味を反映した直接実行アルゴリズムを設定しなければならない．この直接実行アルゴリズムは，言語の構文の認識と意味の解釈実行を一体として記述する実行文法の形[4)]で設計される．

UDEC では，字句解析や構文解析などは，言語の文法をテーブル化してこれを解釈実行する，高速で簡潔なハードウェア機構として実現されている[4),5)]．しかし，最初に試作したモデルである UDEC-I では，意味処理アルゴリズムを系統的に設計する技法が確立していなかったので，意味の解析と実行を行う意味処理部はマイクロプログラム制御で動作するプロセッサ MEGA3[8)] 上で手続き的に実現した．UDEC の2番目のモデルである UDEC-II では，UDEC-I 上でのサンプルプログラムの実行の解析結果に基づいて，意味処理に対する改良が施され，使用される意味手続きを機能的な観点と使用頻度による最適化の観点からプリミティブとして再設計し，アルゴリズムの記述性の改良を行った．さらに，意味処理部のうちから先行制御の可能ないくつかの機能を分離して，独立のモジュールとしてパイプライン構造の各部に分散し，実行の高速化を計ると同時に，これを直接実行アルゴリズムのより形式的な記述に反映させた．

本論文では，形式化した実行文法の形で，PASCAL のサブセット用に設計された直接実行アルゴリズムについて詳述する．

2. UDEC の構成とデータ要素

UDEC-II のハードウェアは，主に，LU (Lexical Unit : 字句解析ユニット)，SU (Scope Unit : 名前の属性検査ユニット)，CU (Control Unit : 制御構造の登録検査ユニット)，SSR (Syntax and Semantics Recognizer : 構文意味認識ユニット)，EA (Effective Address Unit : 有効アドレスの計算ユニット)，EU (Execution Unit : 意味実行ユニット) で構成されている[7),11)]．ここでは，ハードウェアの構成の概略を示し，PASCAL のサブセット[10)]用に設計された意味の解析や実行に必要なデータ要素を，各ユニットごとに定義する．

2.1 字句解析ユニット

字句解析ユニット (LU) には，2つのプログラムカウンタ n-ptr と c-ptr がある．このうち，n-ptr はソースプログラムの文字をプログラムメモリPMから読み出すためのポインタとして使われ，c-ptr はこのユニットで認識された字句レベルのトークンの位置を指すためのポインタとして使われる．c-ptr は，認識されたトークンのコードと型を示す token-code と token-type とともに，SU, CU, SSR などに送られる．一方，ジャンプが起こると，EU から飛び先のア

† Design of the Direct-execution Algorithm for UDEC-II by KOZO ITANO (Institute and Information Sciences and Electronics, University of Tsukuba) and YUTAKA SATO (Computer Science Division, Electrotechnical Laboratory).

本研究は文部省科学研究費補助金・試験研究(1)58850063 および 61850061 によって補助された．

†† 筑波大学電子・情報工学系
††† 電子技術総合研究所ソフトウェア部

ドレスが送られてくるので，これが SSR からの指示にしたがって n-ptr にセットされる．

2.2 名前の属性検査ユニット

手続きディスクリプタとデータディスクリプタのテーブルのうち，連想検索に必要な部分は名前の属性検査ユニット (SU) に置かれる．これらの連想検索部は AT (Associative Table) と呼ばれるテーブルとしてまとめられている．このテーブルの構造を**図1**に示す．AT は，ブロックの識別子と名前のコードの組によって検索され，手続きディスクリプタテーブルとデータディスクリプタテーブル中のディスクリプタへのインデックス pi と di を EA と EU に送る．このとき，AT は，スコープの範囲でもっとも内側の名前が1回の検索で見つけられるように設計されている[7]．この検索は，LU から送られてきたトークンが名前のときだけ実行されるが，このとき同時にトークンの種類がタイプに応じて細分類され，トークンのタイプが置き換えられて後段に送られる．

名前のスコープの制御は，ディスプレイレジスタに基づいて行うが，SU では SSR やEU に対して先行制御を行っているので，ブロックの切り替わるときにブロック識別子が更新されるのを待たなくてはならない．このために，swait と呼ばれる同期のためのフラグが SU 内にあり，BEGIN と END が通過するときに SU の動作をいったん停止し，SSR から指示があるまで動作を行わないようになっている．ブロック識別子 bid の管理のためにスタック sc が使用され，新しい値が設定されるときは，現在の値をスタック sc に保存して，bid の値を更新し，回復はスタックから自動的に行われる．

associative
key field
|←------→|

bid	id	p-type	pi	d-type	di
0	add	FUNC	14	FUNC	28
0	temp	NIL	*	SCALAR	31
...	....	.....	..	.....	..

図 1 AT の構造
Fig. 1 Structure of AT.

associative
key field
|←------→|

bid	loc	type	ci
5	29	WHILE	11
5	36	IF	12
....	....	.....	..

図 2 CT の構造
Fig. 2 Structure of CT.

2.3 制御構造の登録検査ユニット

IF や WHILE などの制御構造のディスクリプタの連想検索部は制御構造解析ユニット (CU) 内に置かれる．この連想検索テーブル CT を**図2**に示す．CT は，制御構造が存在するブロックの識別子と制御構造の先頭のキーワード (IF や WHILE) のアドレスを組にして連想検索され，対応する制御ディスクリプタのインデクスを EU に送る．

この処理は IF や WHILE のトークンが CU を通過するときに行われ，同時に，これらの制御構造がディスクリプタとして完成しているかどうかで，これらのトークンのタイプを置き換えて，SSR 内で展開すべき生成規則を切り替えるようになっている．

2.4 構文意味認識ユニット

構文意味認識ユニット (SSR)[4] には，構文を認識するための基本的なメカニズムとして，生成規則や意味の解析実行の命令コードを保存する文法テーブル，現在展開中の生成規則を指すレジスタ code と，ネストした展開の制御に使う code-stack，意味の実行の抑制[4] を制御する skip-stack がある．

2.5 有効アドレス計算ユニット

名前が与えられたとき，その名前に対応するデータメモリ中のワードのアドレスを計算するために，有効アドレス計算ユニット (EA) は次のようなデータ要素を内部に持つ．まず，データの属性を記述するデータディスクリプタテーブル（図3）には，そのデータのタイプ，サイズ，フレーム中でのオフセット，その名前が存在するネストのレベルが記録されている．実際のメモリ上のアドレスを求めるには，アクティブなフレームのアドレスを管理するディスプレイを用い

る．このユニットからの出力は，データのアドレス xadrs，サイズ xsize，タイプ xtype であり，EU に送られる．

ディスプレイの管理のためにスタック ds が使用され，クロススタックキャッシュ法[9]を用いて管理がされる．したがって，新しい値が設定されるときは，外部のユニット (EU) からデータの受け入れが必要であるが，ディスプレイの値の保存と回復は内部で自動的に行われる．

2.6 意味実行ユニット

意味実行ユニット (EU) は意味の解析と実行を行う主要なユニットである．ディスクリプタテーブルのうち，手続きディスクリプタと制御ディスクリプタの本体は，このユニット内にある PDT と CDT に格納する．ディスクリプタの形式を図 3 に示す．

データの宣言の処理中に，属性を一時的に保存するために使用するレジスタとして，vtyp, vis, vstr, vsize が，また，引数や変数のオフセットの保存用として，numparm, dm-offset, localc がある．アクティブなブロックのうち，頻繁に使用されるもっとも内側のブロックと外側のブロックの識別子を bid と mid として保存する．実行時に使用するスタックは，sn (statement_nest), cl (calling), arp (activation_record_pointer), vl (variable_list), op (operator), vs (value_stack), as (address_stack), ps (parameter stack), tmp (temporary) の計 9 個である．ひとつの意味命令の中でのみ使用される作業用の変数は，p, a, s, v の 4 つを用意する．

procedure descriptor

body-loc	numparm	arsize	level

data descriptor

type	is	size	offset	level

control descriptor for IF

ELSE_loc	exit_loc

control descriptor for WHILE

bool_loc	exit_loc

図 3 ディスクリプタの形式
Fig. 3 Descriptor formats.

3. 意味の解析と実行アルゴリズム

直接実行アルゴリズムの基本的な部分は，実行文法[4]の形で記述されているが，意味解析処理の一部は独立のモジュールとしてパイプライン化されて実行されるので，この部分のアルゴリズムは，実行文法からは分離して設計することにした．以下に，実行文法の形で記述されている言語の構文と意味の解析・実行および，意味解析部のこれとは独立に設計されている部分について説明する．意味解析・実行のアルゴリズムは，Cの文法を基本として記述することにした．これらの記述中で，手続きのパラメータ '-' は，そのパラメータの引渡しがないことを示す．&x は，x のアドレスを渡して値を返してもらうことを意味している．

```
⟨prog⟩      ::=%t{ PROGRAM    ⟨prog-1⟩                          %1 }
⟨prog-1⟩    ::=%t{ ⟨id⟩       ⟨prog-2⟩                       %K+E %2 }
⟨prog-2⟩    ::=%t{ '('        ⟨prog-3⟩                           }
⟨prog-3⟩    ::=%t{ INPUT      ⟨prog-4⟩                           }
⟨prog-4⟩    ::=%t{ ','        ⟨prog-5⟩                           }
⟨prog-5⟩    ::=%t{ OUTPUT     ⟨prog-6⟩                           }
⟨prog-6⟩    ::=%t{ ')'        ⟨prog-7⟩                           }
⟨prog-7⟩    ::=%t{ ';'        ⟨decls⟩ ⟨end-decls⟩                }
⟨end-decls⟩::=%t{ '.'         ⟨dot⟩
                  | ';'       ⟨sub-decls⟩ ⟨end-decls⟩ %K- %3     }
⟨dot⟩       ::=%t{ '.'                                           }
⟨sub-decls⟩::=%t{ PROCEDURE   ⟨p-head-1⟩ ⟨decls⟩
                  | FUNCTION  ⟨f-head-1⟩ ⟨decls⟩
                  | BEGIN     ⟨chk-p⟩                      %A %4 }
⟨chk-p⟩     ::=%r{ 'true'     ⟨stmt-list⟩⟨proc-end⟩        %K+S
                  | 'false'   ⟨stmt-list⟩⟨proc-end⟩              }
⟨proc-end⟩  ::=%t{ END                                     %A %5 }
⟨p-head-1⟩  ::=%t{ ⟨id⟩       ⟨form-args⟩ ⟨f-head-4⟩         %6 }
⟨f-head-1⟩  ::=%t{ ⟨id⟩       ⟨form-args⟩ ⟨f-head-3⟩         %7 }
⟨f-head-3⟩  ::=%t{ ':'        ⟨f-type⟩ ⟨f-head-4⟩                }
⟨f-head-4⟩  ::=%t{ ';'                                           }
```

```
%1 : bid = 0; lev = 0; push_display(lev,-,bid);
%2 : pi = new_at(bid,token_code,PROC,-);
     new_pdt(pi)(-,-,-,lev);
     lev++; bid = pi; push_display(lev,-,bid);
%3 : pop_tmp(&bid);
%4 : put_pdt(pi)(c_ptr,-,-,-);
     if(mid!=bid) result(TRUE); else result(FALSE);
%5 : lev--; pop_display();
%6 : pi = new_at(bid,token_code,PROC,-);
     new_pdt(pi)(-,-,-,lev); lev++; push_tmp(bid);
     bid = pi; push_display(lev,-,bid);
%7 : pi = new_at(bid,token_code,FUNC,FUNC);
     new_pdt(pi)(-,-,-,lev); new_ddt(di)(-,0,0,0,lev);
     lev++; push_tmp(bid); push_tmp(di); bid = pi;
     push_display(lev,-,bid);
```

図 4 手続きの宣言
Fig. 4 Procedure declarations.

実際の意味処理は，マイクロプログラムやハードウェアで実現されているが，ここでは全体を見通しよく示すために抽象度の高い形式でアルゴリズムを示すことにした．

3.1 宣言の処理

宣言の処理には，手続きの構造の解析とデータの宣言の解析がある．

(1) 手続きの宣言

図4 に手続きの宣言を処理する部分の実行文法を示す．生成規則の右辺の最初の %t と %r は，SSR が入力としてプログラムの記号をとるか，EU における意味解析または意味実行の結果をとるかの選択の指定である．%K+E, %K+S, %K− は実行の抑制を行うためのモードを管理するスタック (skip_stack) を制御する指令である．%K+E と %K+S は，skip_stack に 'exec' と 'skip' をそれぞれプッシュする．%K- は skip_stack をポップする．skip_stack のトップが 'exec' のときだけ意味の実行が許される．%A は先行制御されている名前に関する意味解析を再開させるための指示で，SU との間の同期に使用される．生成規則の最後にある %数字は，EUに送られて実行される意味解析や意味実行の命令コードである．

ここで関係している意味命令が図に示してある．各意味命令の記述の中で使われている手続きは，意味の解析や実行のためのプリミティブであり，UDEC のマイクロルーチンとみなすことができる．

・ディスプレイの制御

SU中のブロック識別子およびEA中のディスプレイを更新するための処理を含む．手続き push_display(lev, ptr, bid) は，SU 中のブロック識別子に bid を設定し，EA中のディスプレイのレベル lev にフレームのポインタ ptr を設定する．また，手続き pop_display はこれらブロック識別子とディスプレイを元の状態に回復することを指示するために使用する．

・スタックの制御

各スタックへのデータのプッシュとポップを指示する手続きである．push_x と pop_x で x はスタックの名前を指定する．データは引数の形で書く．

・ATの登録

名前の連想テーブルへのエントリの登録は，手続き new_at(bid, name, flg) で行う．引数は，bid がブロック識別子，name が名前，flg が手続きかデータかを示すフラグである．この手続きは，結果として PDT か DDT 中のインデックスを返す．

・PDT の登録と追記

手続きディスクリプタのテーブルへの書き込みの手続きは，登録用の new_pdt と追記用の put_pdt が

```
⟨decls⟩     ::=%t{ VAR        ⟨decl-list⟩ ⟨sub-decls⟩      %8
                 | PROCEDURE  ⟨p-head-1⟩ ⟨decls⟩
                 | FUNCTION   ⟨f-head-1⟩ ⟨decls⟩
                 | BEGIN      ⟨chk-p⟩                  %A  %4  }
⟨decl-list⟩::=%t{ ⟨id⟩+⟨id0⟩+⟨id1⟩+⟨id2⟩+⟨id3⟩+⟨id4⟩
                              ⟨id-list-1⟩ ⟨decl-lst2  ⟩  %9  }
⟨decl-lst2⟩::=%t{ ';'        ⟨more-decl⟩                      }
⟨more-decl⟩::=%t{ ⟨id⟩+⟨id0⟩+⟨id1⟩+⟨id2⟩+⟨id3⟩+⟨id4⟩
                              ⟨id-list-1⟩ ⟨decl-lst2⟩     %9
                 | PROCEDURE + FUNCTION + BEGIN      %L-  %10 }
⟨id-list⟩   ::=%t{ ⟨id⟩+⟨id0⟩+⟨id1⟩+⟨id2⟩+⟨id3⟩+⟨id4⟩
                              ⟨id-list-1⟩                 %9  }
⟨id-list-1⟩::=%t{ ','        ⟨id-list⟩
                 | ':'        ⟨type⟩                           }
⟨type⟩      ::=%t{ INTEGER + REAL                          %11
                 | ARRAY      ⟨array-1⟩                   %12 }
⟨s-type⟩    ::=%t{ INTEGER + REAL                          %13 }
⟨f-type⟩    ::=%t{ INTEGER + REAL                          %14 }
⟨array-1⟩   ::=%t{ '['        ⟨array-2⟩                        }
⟨array-2⟩   ::=%t{ ⟨u-int⟩    ⟨array-3⟩                   %15 }
⟨array-3⟩   ::=%t{ '..'       ⟨array-4⟩                        }
⟨array-4⟩   ::=%t{ ⟨u-int⟩    ⟨array-5⟩                   %16 }
⟨array-5⟩   ::=%t{ ']'        ⟨array-6⟩                        }
⟨array-6⟩   ::=%t{ OF         ⟨s-type⟩                         }
⟨form-args⟩::=%t{ '('         ⟨parm-list⟩                 %17
                 | ':'+';'                               %L- %17 }
⟨parm-list⟩::=%t{ VAR         ⟨id-list⟩ ⟨more-parm⟩            }
⟨more-parm⟩::=%t{ ';'         ⟨parm-list⟩
                 | ')'                                       %18 }
```

```
%8  : vis = 0; dmoffset = 1;
%9  : if(check_dup_id(token_code)) error();
      push_vl(token_code);
%10 : top_arp(&p); put_pdt(pi)(-,-,-,p,-);
%11 : vstr = SCALAR; vsize = 1;
%12 : vstr = ARRAY;
%13 : vtype = token_type; pop_vl(&p)
      while(p!=NIL/*notempty*/) { di = new_at(bid,p,-,vstr);
       new_ddt(di)(vtype,vis,vsize,dmoffset,lev);
       dmoffset = dmoffset + vsize; pop_vl(&p);}
%14 : pop_tmp(&p); put_ddt(p)(token_code,-,-,-);
%15 : if(token_code!=0) error();
%16 : vsize = token_code + 1;
%17 : dmoffset = 0; numparm = 0; vstr = SCALAR;
      vsize = 1; vis = 1;
%18 : put_pdt(bid)(-,numparm,-,-);
```

図5 データの宣言
Fig. 5 Data declarations.

ある．いずれの場合も，AT の検索の結果得られたインデックス pi を用いて，put_pdt(pi) (flg, loc, parm, frame) のようにディスクリプタにアクセスする．この場合は，flg が関数か手続きかを示すフラグ，loc が手続きまたは関数の本体の位置，parm がパラメータの数，frame がフレームの大きさを示すパラメータである．

(2) データ宣言

図5 にデータの宣言を実行するための実行文法を示す．生成規則の中で新しい記法は，%L- である．これは，字句解析ユニットに対して，最新のトークンを再送することを要求する指示である．意味命令の記述の中で，連想テーブルATへのエントリの書き込みは，手続き名と同様に new_at で行い，DDT へのディスクリプタの書き込みは，new_ddt と put_ddt で行う．これらの手続き new/put_ddt(type, is, str, size, offset) の引数は5つあり，type はデータのタイプ，is は引数かどうかのフラグ，str はデータの構造，size はデータの大きさ，offset はフレームの中での偏位を表す．

ここで，実行の対象にしているPASCALのデータの宣言では，データの名前のリストの後に型がくるので，名前をいったん vl に保存してから，型を含む属性を vtype, vis, vstr, vsize, voffset の5つのレジスタに作成し，ディスクリプタの作成に移る．この際，データの位置を計算するためのレジスタとして dmoffset を，また，引数の数を計算するレジスタとして numparm を使用する．その他の手続きとしては，error と check_dup_id があるが，前者はエラーの報告に，後者は名前の重複定義の検査に用いられる．

3.2 ステートメントの識別法

ステートメントの種類は，構文レベルでは，生成規則の1回の展開で識別される．この部分に対応する実行文法を**図6**に示す．ここで，〈stmt〉の右辺の終端記号として示されている記号は，BEGIN を除いて，SSR の前段にある SU と CU において意味解析を行って細分類されたトークンに対応している．これらのトークンは次のようなケースを示す．

〈id 0〉：単純変数である．

```
〈comp-stmt〉::=%t{ BEGIN      〈stmt-list〉 〈end〉                    }
〈stmt-list〉::= {             〈stmt〉 〈stmt-lstl〉                   }
〈stmt〉     ::=%t{ 〈IF-f〉     ...
                 | 〈IF-s〉     ...
                 | 〈WHILE-f〉  ...
                 | 〈WHILE-s〉  ...
                 | BEGIN      〈stmt-list〉 〈end〉    %A  %K+S
                 | 〈id0〉+〈id2〉...
                 | 〈id1〉      ...
                 | 〈id3〉      ...                                  }
〈stmt-lstl〉::=%t{ ';'        〈stmt-list〉
                 | END                                %L- %K-     }
〈end〉      ::=%t{ END                                  %A          }
```

図6 ステートメント
Fig. 6 Statements.

```
〈stmt〉     ::=%t{ .......
                 | 〈id3〉      〈act-args〉 〈proc_call〉      %19
                 | .......
〈proc_call〉::=%k{ 'exec'     〈comp-stmt〉 〈pcretn1〉       %20
                 | 'skip'                                        }
〈act-args〉 ::= {             〈act_arg2〉                  %21 }
〈act_arg2〉 ::=%r{ 'true'     〈act_arg3〉
                 | 'false'    〈act_arg4〉                        }
〈act_arg3〉 ::=%t{ DIV + MOD + '/' + AND + '+' + '-'
                   OR  + '=' + '<>'+ '>' + '>='+ '<'
                   '<='+ ')' + ']' + THEN+ DO  + END
                   ELSE+ ';'                          %L-      }
〈act_arg4〉 ::=%t{ '('        〈variable〉 〈more-args〉      %22 }
〈more-args〉::=%t{ ','        〈variable〉 〈more-args〉      %23
                 | ')'                                       %24 }
〈func-call〉::=%k{ 'exec'     〈comp-stmt〉 〈fvretn〉   %Lc  %20
                 | 'skip'                                        }
〈fvretn〉   ::= {             〈pcretn〉                    %25 }
〈pcretn〉   ::= {             〈pcretn-2〉                  %26 }
〈pcretn-2〉 ::= {                                          %Lc %27 }
```

```
%19 : push_tmp(pi);
      get_pdt(pi)(-,-,&numparm,-,-);
%20 : push_ps(numparm); push_cl(n_ptr,bid); /* call */
      pop_tmp(&pi); bid = pi;
      get_pdt(pi)(-,&p,-,&v,&a);
      push_arp(carp); carp = narp; narp = carp + v;
      push_display(a,carp,bid);
      setjump(p);
%21 : if(numparm==0) result(TRUE); else result(FALSE);
%22 : localc = 1;
%23 : pop_as(&a,-,&p); push_ps(a,p); localc++;
%24 : pop_as(&a,-,&p); push_ps(a,p);
      if(localc!=numparm) error();
%25 : /* set func value */ pop_vs(&p,-); v = dm(carp);
      push_vs(p,v);
%26 : narp = carp; pop_arp(&carp); pop_ps(&numparm);
%27 : /* return */ pop_display(); pop_cl(&p,&bid);
      setjump(p);
```

図7 手続きの呼び出し
Fig. 7 Procedure calls.

〈id1〉: アレイである.
〈id 2〉: 関数の中のリターン値が代入される名前である.
〈id 3〉: 手続き名である.
〈id 4〉: 関数名である.
〈if-f〉: この IF 文に対応する制御ディスクリプタが存在しないか, または, 未完成である.
〈if-s〉: この IF 文に対応する制御ディスクリプタが既に存在している.
〈while-f〉: この WHILE 文に対応する制御ディスクリプタが存在しないか, または, 未完成である.
〈while-s〉: この WHILE 文に対応する制御ディスクリプタが既に存在している.

3.3 手続きの呼び出し

手続きと関数の呼び出しに関わる部分の実行文法を**図7**に示す. この部分は, 手続きまたは関数の呼び出しの前処理, 実引数の処理, 呼び出し, 復帰に分かれており, 関数の場合は復帰のときに関数値を設定する処理が加わっている. 生成規則部の記述の中で新たに現れるものは, %k と %Lc である. %k はスキップモードの値で以後の流れを切り替えるためのものである. %Lc はジャンプなどによるプログラムカウンタの変更に伴って, LUで先行制御されている字句解析の結果をキャンセルするための指示である.

意味実行命令の中で使われている手続きのうち, result は, EU から SSR に意味解析や実行の結果を送るためのもので, 引数で指定された値が送られる. setjump は, EU から LU に対して新しいプログラムカウンタの値を送るためのものである.

使用されているデータ要素の中で, numparm は手続きまたは関数の仮引数の数を保存し, localc は実引数の数をカウントするのに使われている. 引数はスタック ps に積んで渡される.

3.4 制御構造の認識と実行

ここで取り扱う制御構造は IF と WHILE である. これらの制御構造は, 解析しながら実行される.

(1) IF ステートメント

図8に IF ステートメントの実行文法を示す. IF ステートメントは, つぎのような構造を持つ.

```
IF expression THEN statement ELSE statement
^                            ^              ^
a                            b              c
```

ここでは, 3つの位置 a, b, c が重要な意味を持つ. a は, この IF ステートメントの識別子として使う. b は, ELSE の位置, c はこの IF ステートメントのつぎのステートメントの位置を示す.

IF ステートメントの実行は, まず, 最初のキーワードである IF を認識することから始まる. これは, CU で行われ, a をキーとして, 連想テーブル CT が検索される. この結果, 対応するディスクリプタが完成しているかどうかが判定され, トークンのタイプが〈if-f〉か〈if-s〉かに置き換えられる. SSR では, こ

```
〈stmt〉     ::=%t{ .....
                 | 〈IF-f〉    〈if-f〉                    %28
                 | 〈IF-s〉    〈if-s〉                    %29
                 | .....
〈if-f〉     ::=  {           〈b-expr〉〈f-if-2〉                }
〈if-s〉     ::=  {           〈b-expr〉〈s-if-2〉                }
〈f-if-2〉   ::=%t{ THEN      〈f-if-3〉                  %30 }
〈f-if-3〉   ::=%r{ 'true'    〈stmt〉〈t-else〉    %A %K+S
                 | 'false'   〈stmt〉〈f-else〉      %K+S      }
〈t-else〉   ::=%t{ ELSE      〈stmt〉〈t-else-1〉        %31
                 | ';' + END                    %L-  %32 }
〈f-else〉   ::=%t{ ELSE      〈stmt〉〈else-2〉      %K-  %33
                 | ';' + END                %K- %L-  %34 }
〈t-else-1〉 ::=  {                          %A %K-       }
〈else-2〉   ::=  {                                  %35 }
〈if-s〉     ::=%t{ THEN      〈s-if-3〉                  %30 }
〈s-if-3〉   ::=%r{ 'true'    〈t-s-if〉
                 | 'false'   〈f-s-if〉              %Lc  %36 }
〈t-s-if〉   ::=  {           〈stmt〉〈s-if-4〉               }
〈f-s-if〉   ::=%r{ 'true'
                 | 'false'   〈stmt〉〈s-if-4〉               }
〈s-if-4〉   ::=  {                          %Lc  %37 }
```

```
%28 : ci = new_ct(bid,cptr);
      push_sn(ci); new_cdt[ci](-,NIL);
%29 : push_sn(ci);
%30 : pop_vs(&p,&r); if(p!=BOOL) error(); pop_op(-);
      result(r);
%31 : put_cdt[ci](n_ptr,-);
%32 : put_cdt[ci](NIL,c_ptr); pop_sn(&ci);
%33 : put_cdt[ci](n_ptr,-);
%34 : put_cdt[ci](NIL,n_ptr); pop_sn(&ci);
%35 : put_cdt[ci](-,n_ptr);   pop_sn(&ci);
%36 : get_cdt[ci](&p,&a);
      if(p==NIL) { setjump(a); pop_sn(&ci); result(TRUE); }
      else       { setjump(p);              result(FALSE); }
%37 : get_cdt[ci](-,&p); setjump(p); pop_sn(&ci);
```

図 8 IF 文
Fig. 8 If statements.

れをトークンとして受け入れ，これ以降の展開を切り替えて実行すべき意味命令のシークエンスを変更する．このような静的な意味処理とともに，条件式の評価によって THEN 以下と ELSE 以下の実行を切り替えるような動的な処理も同様に実行する．

意味命令の中で，new_ct(a, b) は，対応する制御構造に対するエントリがまだ存在しないときに，新たにエントリを作成して登録することを，EU から CT に対して指示するための手続きである．new_cdt〔i〕(a, b) は，対応する制御ディスクリプタを新たに作成してインデックス i で示される位置に登録する．put_cdt と get_cdt はディスクリプタの更新と読み出しに使用する．

（2） WHILE ステートメント

図 9 に WHILE ステートメントの実行文法を示す．WHILE ステートメントは，つぎのような構造を持つ．

```
WHILE expression DO statement
^       ^                  ^
a       b                  c
```

a はこの WHILE ステートメントの識別子として使う．b は条件式の位置，c はこのステートメントのつぎのステートメントの位置を示す．

WHILE ステートメントの実行は，まず，最初のキーワードである WHILE を認識することから始まる．これは，CU で行われ，a をキーとして，連想テーブル CT が検索される．この結果，対応するディスクリプタが完成しているかどうかが判定され，トークンのタイプが〈while-f〉か〈while-s〉かに置き換えられる．SSR では，これをトークンとして受け入れ，IF と同様に実行する．

```
〈stmt〉     ::=%t{ ......
                 | 〈WHILE-f〉 〈while-f〉                    %38
                 | 〈WHILE-s〉 〈while-s〉                    %29
                 | ......
〈while-f〉  ::=  {          〈b-expr〉 〈f-while-2〉              }
〈while-s〉  ::=  {          〈b-expr〉 〈s-while-2〉              }
〈f-while-2〉::=%t{ DO       〈f-do〉                       %30 }
〈f-do〉      ::=%r{ 'true'  〈stmt〉 〈f-while-3〉
                 | 'false'  〈stmt〉 〈f-while-5〉   %K+S       }
〈f-while-3〉::=  {          〈b-expr〉 〈s-while-2〉  %Lc  %39 }
〈f-while-5〉::=  {                                 %40 }
〈s-while-2〉::=%t{ DO       〈s-do〉                       %30 }
〈s-do〉      ::=%r{ 'true'  〈stmt〉 〈s-while-3〉
                 | 'false'  〈s-while-5〉                   }
〈s-while-3〉::=  {          〈b-expr〉 〈s-while-2〉  %Lc  %41 }
〈s-while-5〉::=  {                                 %Lc  %42 }
```

```
%38 : ci = new_ct(bid,c_ptr);
      push_sn(ci); new_cdt[ci](n_ptr,NIL);
%39 : put_cdt[ci](-,n_ptr);
      get_cdt[ci](&p,-); setjump(p);
%40 : put_cdt[ci](-,n_ptr); pop_sn(&ci);
%41 : get_cdt[ci](&p,-); setjump(p);
%42 : get_cdt[ci](-,&p); setjump(p); pop_sn(&ci);
```

図 9 WHILE 文
Fig. 9 While statements.

3.5 式の評価

図 10 に式の評価を行うための実行文法を示す．式の評価は左から順に走査しながらスタックを用いて行う．オペレータの優先順位は，文法のレベルでは区別しないで，優先順位を記述したテーブルを参照しながら行う．オペレータと値は別々のスタック op と vs に積み，評価は，手続き eval 2 か eval を呼んで実行する．eval 2 では，スタックトップのオペレータと現在のオペレータの優先順位を比較して，現在のオペレータの優先順位が高いときは評価を停止する．eval では，':=' や '(' がオペレータとして現れるまで，無条件に評価を繰り返す．

3.6 データの転送

図 11 にデータの転送に関わる部分の実行文法を示す．この部分では，主に，アドレスの計算，データのフェッチ，代入について処理している．

（1） アドレスの計算

メモリ上のデータのアドレスの計算は，EA の内部で，フレームのアドレスを管理するディスプレイ (fr_display) とデータディスクリプタを用いて行う．すなわち，データの宣言のレベル (level) とオフセット (offset) をデータディスクリプタから知り，

xadrs=fr_display〔level〕+ offset

として計算し，配列の場合はデータのサイズ (xsize) とタイプ (xtype) を加えて EU へ送る．EU では，これらのデータ (xadrs, xtype, xsize) をアドレススタック (as) にセットする．

（2） データのフェッチ

データのフェッチは，基本的に，アドレススタック (as) のトップにあるアドレスでデータメモリ (dm) を読みだしてバリュースタック (vs) にプッシュすることである．このとき，アドレススタック (as) は自動的にポップされる．このため，場合によっては，EA から得たアドレスをスタックに積まないで，直接にデータをフェッチ

```
⟨b-expr⟩   ::= {          ⟨expr⟩                          %43 }
⟨expr⟩     ::=%t{ '+' + '-'  ⟨factor⟩ ⟨expr-1⟩             %44
                | ⟨id0⟩+⟨id2⟩⟨d-fetch⟩⟨expr-1⟩
                | ⟨id1⟩      ⟨subscript⟩ ⟨fac-exp-1⟩      %45
                | ⟨id4⟩      ⟨act-args⟩ ⟨f_call⟩          %46
                | ⟨u-const⟩  ⟨expr-1⟩                     %47
                | '('        ⟨expr⟩ ⟨expr-2⟩              %48
                | NOT        ⟨factor⟩ ⟨expr-1⟩                 }
⟨fac-exp-1⟩::= {          ⟨factor-1⟩ ⟨expr-1⟩               }
⟨expr-2⟩   ::=%t{ ')'     ⟨expr-1⟩                          }
⟨f-call⟩   ::= {          ⟨func-call⟩ ⟨expr-1⟩              }
⟨expr-1⟩   ::=%t{ DIV + MOD + '/' + '*' + AND + '+' +
                  '-' + OR   ⟨factor⟩ ⟨expr-1⟩            %49
                | '=' + '⟨⟩' + '⟩' + '⟩=' + '⟨' + '⟨='
                           ⟨expr⟩                         %49
                | ')'                                  %L.- %50
                | ']'                                  %L.- %51
                | DO + THEN + ELSE + END + ';'         %L.- %52 }
⟨factor⟩   ::=%t{ ⟨id0⟩+⟨id2⟩⟨d-fetch⟩
                | ⟨id1⟩      ⟨subscript⟩ ⟨factor-1⟩       %45
                | ⟨id4⟩      ⟨act-args⟩ ⟨func_call⟩       %46
                | ⟨u-const⟩                               %47
                | '('        ⟨expr⟩ ⟨cls-paren⟩           %48
                | NOT        ⟨factor⟩                     %53 }
⟨factor-1⟩ ::= {                                         %54 }
⟨cls-paren⟩::=%t{ ')'                                        }
⟨variable⟩ ::=%t{ ⟨id0⟩ + ⟨id2⟩                          %45
                  ⟨id1⟩      ⟨subscript⟩                 %45 }
⟨subscript⟩::=%t{ '['      ⟨expr⟩ ⟨cls-brakt⟩           %55 }
⟨cls-brakt⟩::=%t{ ']       ⟨adrs-cal⟩                        }
-----------------------------------------------------------------
%43 : push_op(ASGN);
%44 : push_op(token_type); push_vs(INT,0);
%45 : push_as(xadrs,xsize,xtype); /* set address */
%46 : push_tmp(pi); push_vs(xtype,0);
      get_pdt[pi](-,-,&numparm,-,-);
%47 : push_vs(token_type,token_code);
%48 : push_op(token_type);
%49 : opensymbol = ASGN;   eval2(); push_op(token_type);
%50 : opensymbol = LPAREN; eval();
%51 : opensymbol = LBRACK; eval(); pop_op(-);
%52 : opensymbol = ASGN;   eval();
%53 : push_op(token_code);
%54 : /* data fetch */ pop_as(&a,-,&p); a = dm[a];
      push_vs(p,a);
%55 : push_op(token_type);
```

図 10 式の評価
Fig. 10 Evaluation of expressions.

する命令も用意している.

(3) 代　　入

変数への値の代入は，まず，代入文の左辺にある名前を実行した時点でこのアドレスをアドレススタック (as) に積み，右辺の式を評価して得た結果の値が積まれているバリュースタック (vs) から値を取り出して，アドレススタック (as) のトップにあるアドレスへ書き込みを行う.

4. 考　　察

アルゴリズムの設計にあたって，問題となった点のうちのいくつかについて考察する.

4.1 ユニット間の通信量の減少

全体の実行アルゴリズムを，物理的に独立のユニットに分散させるためには，データの参照が局所的になるように設計しなければならない．しかし，あらゆるデータの参照を局所的にすることは，現実には困難であった．例えば，SU 中にある連想テーブル AT の大部分の参照は，SU 内で LU から送られてきたトークンを検索するためのものであるが，AT のエントリを作成するための情報は EU 中にあり，EU からも AT がアクセスできなくてはならない.

このように，あるデータ要素が2つ以上のユニットからアクセスされるときに，どのユニットにそのデータ要素を配置するかは，ユニット間の通信を最適化して，全体の性能を上げることにつながるので重要な問題である．そこで，つぎのような方針で設計を行った.

(a) 実行頻度の高い処理が1つのユニット内に局所化するようにする.
(b) ユニット間の通信を伴うような処理の場合はなるべく，通信に必要な情報の量が少なくなるように設計する.

実際に，AT, CT, PDT, CDT, DDT, PM, DM などの主要なデータ要素は，これらの条件を考察してそれぞれのユニットに配置した．その他のレジスタなどについては，本来1つのものを分割して関係のあるユニットに分散させたり，コピーを作って分散させるなどの技法を用いた.

4.2 パイプライン中のデータの流れの平滑化

通信の情報量とともに問題となるのが，ユニット間の同期のために，処理がブロックされてパイプラインの効率が損なわれることによる時間損失である．字句解析，構文解析，意味解析という分類は，もともとパイプライン化するのに自然であり UDEC でも採用しているが，より細かく分析すると，意味解析が構文解析の後にあるためにパイプラインの流れが乱れている部分がいくつかある．これらのうち，処理の頻度の高いものは改善の必要がある．たとえば，UDEC-I ではあらゆる意味解析を EU で行っていたので，SSR

```
⟨stmt⟩      ::=%t{ .....
                 | ⟨id0⟩+⟨id2⟩⟨asg-stmt1⟩                    %56
                 | ⟨id1⟩      ⟨subscript⟩ ⟨asg-stmt1 ⟩      %56
                 | .....
⟨asg-stmt1⟩::=%t{ ':='        ⟨expr⟩ ⟨asg-stmt2 ⟩           %57 }
⟨asg-stmt2⟩::=  {                                           %58 }
⟨adrs-cal⟩  ::=%k{ 'exec'                                   %59
                 | 'skip'                                   %60 }
⟨d-fetch⟩   ::=%k{ 'exec'                                   %61
                 | 'skip'                                   %62 }
-------------------------------------------------------------
%56 : push_as(xadrs,xsize,xtype); /* address */
%57 : push_op(ASGN);
%58 : pop_as(&a,-,&p); pop_vs(&s,&v); if(p!=s) error();
      dm[a] = v; pop_op(-);   /* assign */
%59 : pop_vs(&p,&v); if(p!=INT) error(); pop_as(&a,&s,&p);
      if(v)s) error(); a = a + v; push_as(a,s,p);
%60 : pop_vs(&p,-); if(p!=INT) error();
%61 : v = dm[xadrs]; push_vs(xtype,v);
%62 : push-vs(xtype,0);
```

図 11 データの転送
Fig. 11 Data transfer.

で名前を認識するためには，SSR の後段にある EU で意味の解析を行ってこの結果を SSR にフィードバックしてやることが必要であった．しかし，これでは，パイプラインの中のデータの流れがブロックしてしまう．

そこで，データの流れがなるべくスムーズになるようにアルゴリズムを再設計した．これはまた，負荷の重い EU の処理[4),7)]を部分的に先行処理させることによって，パイプラインの負荷バランスを改善することにつながっている．このうちの主要なものを次に示す．

（1） SU による名前の細分類の先行処理

名前のスコープに関する規約は，手続き型の言語においては共通しており，ブロックの切れ目を BEGIN や END などで仮に認識することができる．したがって，比較的単純なメカニズムでスコープルールに基づいた名前の属性の検査が可能であり，名前の属性を構文解析以前に行うようにアルゴリズムを変更した．名前の検索は，時間のかかる処理であるから，先行処理することで全体の処理時間を短縮するのに効果がある．

（2） CU による制御構造の先行認識

現在のアルゴリズムでは，IF や WHILE の制御構造は実行時に抽出されてディスクリプタが作られるが，このために制御構造に対応するディスクリプタが既に作られているかどうかで生成規則を切り替えている．この検査をするには，ディスクリプタの検索をすることが必要であり，先行処理を行うことが効果的である．また，IF や WHILE などのトークンが認識できるだけでよいので構文解析を行う前に実行することが困難なくできる．

（3） EAによる有効アドレスの計算の先行処理

データメモリ中のデータの実アドレスを求める処理は，EU での演算やデータの転送に先だって行うことが必要であるが，名前の属性検査が可能であるのと同様の理由により，アドレスの計算を先行処理することができる．このアドレスの計算処理自体も負荷の重い処理であり，先行処理をすることが効果的である．

4.3 ユニット間の同期

ユニットの同期のうち，データの授受に関する物理的な同期は，ハードウェアが自動的に制御しているので問題ない．ここで，注意しておく必要があるのは，ユニット間の論理的なレベルでの同期である．

（1） ジャンプ発生時の LU の再同期

LU と他のユニットの間の同期は，主に，制御の流れにジャンプが起こるとき必要になる．このとき，先行処理はすべて無駄になり，キャンセルしなくてはならない．LU，SSR，EU などのユニットは，自立しているのでメッセージのレベルで同期をとる必要があるが，これは次のようにして解決されている．

ジャンプが起こるときは，まず，SSR は %Lc によって SSR 自体が入力を受け付けないで読み飛ばすモードに移行する．このあとで，EU から LU に対して setjump 手続きによりジャンプアドレスが送られる．LU は，ジャンプアドレスを受け取ると，それまでの処理をキャンセルして，新しいアドレスから字句解析を始め，最初にこのことを示すコントロール用のトークンを付けて SSR に送る．SSR では，このトークンを受け取ると，またもとの状態に復帰して構文の認識を再開する．

（2） ブロックの切り替えに伴う SU の再同期

SU と他のユニットの間の同期は，主に，名前のスコープの変更を伴うブロックの変わり目で起こる．すなわち，ブロックが変更されたら，ブロック識別子を変更するまで次の名前を受け付けてはならないので，このための同期のメカニズムが必要である．このためには，本来，ブロックの正確な始まりと終わりが認識できる必要があるが，構文解析以前にこれを行うの

は困難である．そこで，前述のように，BEGIN と END のトークンが検出されたとき，ブロックが切り替わっていると仮定して，このとき，SU 内にある swait フラグを立て，これを EU がリセットしてくれるまで処理を中断している．この方法では，ブロックの境目以外のところでも，待ち合わせをすることになるが，この方法を採用することで 4.2(1)，(3)などが実現されている．

5. おわりに

直接実行型計算機 UDEC-II のために開発された，PASCAL のサブセットの直接実行アルゴリズムを，実行文法の形で形式化して示した．本論文で示したアルゴリズムは，パイプライン処理向きにきめ細かく改良されており，UDEC-II のハードウェアの構成を反映している．このアルゴリズムは，既にハードウェア上に実装されて動作しており，性能の解析も行われている[7]．また，ここで示した直接実行アルゴリズムは，マイクロプログラムの形で設計されているものや，ハードウェアで実現されているものを，形式的に書き直したものである．本来は，このようなレベルの記述から自動的に実際の制御アルゴリズムが生成できるようなツールが存在することが望ましいので，今後の課題としたい．

参考文献

1) Itano, K.: PASDEC: A Pascal Interactive Direct-Execution Computer, *Proc. of High-Level Language Computer Architecture Conference*, pp. 152-169 (1982).
2) 板野肯三，佐藤　豊：汎用直接実行型計算機 UDEC のアーキテクチャ，情報処理学会論文誌，Vol. 27, No. 8, pp. 747-753 (1986).
3) Itano, K. and Sato, Y.: Architecture of the Universal Direct-Execution Computer UDEC, *Proc. of Hawaii International Conference on System Sciences*, pp. 264-273 (1987).
4) 板野肯三，佐藤　豊，林　謙治：UDEC における直接実行アルゴリズムの記述法とその評価，情報処理学会論文誌，Vol. 27, No. 11, pp. 1106-1111 (1986).
5) 板野肯三，佐々木日出美，山形朝義：連想記憶に基づくパイプライン型文字列検索アルゴリズム情報処理学会論文誌，Vol. 26, No. 6, pp. 1152-1155 (1985).
6) 板野肯三，佐藤　豊，山形朝義：パイプライン型字句解析プロセッサの設計と実現，情報処理学会論文誌，Vol. 28, No. 1, pp. 82-90 (1987).
7) 板野肯三，佐藤　豊，杉原敏昭：直接実行型計算機 UDEC の意味実行部の設計と評価，テクニカルノート HLLA-176，筑波大学電子情報工学系(1986).
8) 板野肯三，杉原敏昭：マイクロプログラマブルプロセッサ MEGA 3 の設計，テクニカルノート HLLA-41，筑波大学電子情報工学系 (1984).
9) 板野肯三，佐藤　豊，中村敦司：クロススタックキャッシュを用いたブロック構造言語のためのアドレッシング機構，情報処理学会論文誌，Vol. 27, No. 9, pp. 916-920 (1986).
10) Aho, A. and Ullman, J.: Principles of Compiler Design, Addison-Wesley, Reading, MA (1979).
11) 板野肯三，佐藤　豊，杉原敏昭：直接実行型計算機 UDEC のハードウェア構成と性能評価，テクニカルノート HLLA-188，筑波大学電子情報工学系 (1987).

(昭和 62 年 5 月 29 日受付)
(昭和 62 年 9 月 9 日採録)

板野　肯三（正会員）

昭和 23 年生．昭和 46 年東京大学理学部物理学科卒業．昭和 48 年同大学大学院修士課程修了．昭和 51 年同博士課程単位取得後退学．理学博士．筑波大学計算機センタ準研究員，同大学電子・情報工学系助手，講師を経て，現在，同助教授．コンピュータアーキテクチャ，オペレーティングシステム，プログラミング言語処理系に興味を持つ．ソフトウェア科学会，IEEE, ACM 各会員．

佐藤　　豊（正会員）

昭和 35 年生．昭和 57 年筑波大学第三学群情報学類卒業．昭和 59 年同大学院修士課程理工学研究科修了．昭和 62 年同大学院博士課程工学研究科修了．工学博士．同年電子技術総合研究所入所．在学中は，プログラミング・システムのユーザ・インタフェースおよび構成法の研究に従事．ソフトウェア科学会会員．