Vol. 27 No. 11 情報処理学会論文誌 Nov. 1986

UDEC における直接実行アルゴリズムの記述法とその評価†

板野 肯三†† 佐藤 豊††† 林 謙治††††

ソースプログラムを直接実行するアルゴリズムを記述する文法として，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設計し，この実行文法を高速に実行するハードウェアとして構文意味認識機構を実現した．実行文法はこのハードウェア中の制御テーブルに格納され，このテーブルの書き換えによって対象言語が切り換えられる．本方式の評価のために，PASCAL のサブセットの直接実行アルゴリズムを実行文法で記述し，記述の規模や実行時の性能を評価した．

1. まえがき

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，汎用の直接実行型計算機 UDEC を試作した[1)~3)]．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で設計した．

本論文ではこのうち，直接実行アルゴリズムを記述する実行文法と，これをハードウェアによって高速に実行する構文意味認識機構[3)]について詳述する．以下では，まず直接実行の基本方針について簡単に述べ，実行文法と PASCAL のサブセットの記述例を説明し，構文意味認識機構のハードウェア構成，実現された実行アルゴリズムの規模および性能等について述べる．

2. 直接実行の基本方針

UDEC ではソースプログラムを，解析のできる最小単位ごとに直接実行する．このために，文字列としてのプログラムを読んで字句解析されたトークンごとに，構文を解析し，意味を解析し，実行する．このうち，解析と実行は分離せず一体的な処理として実現するが，不必要な解析の繰り返しは抑制して高速化を計り，解析だけを行うことが必要な場合には実行を抑制する．

（1） 宣言部の処理

データや手続きの宣言部は，この部分が走査されるときに一度だけ解析する．この解析では手続きやデータ構造を記述するディスクリプタを作成し，ブロック構造の識別子や名前とデータ型やアドレス情報などを組として格納する．以降の実行では名前からこのディスクリプタを連想的に検索して取り出し，手続きやデータの実体にアクセスする．作成されたディスクリプタはプログラムが変更されない限り消滅しない．

（2） 制御構造の抽出

if や while などの制御文は，それらを最初に実行するときに解析して構造を抽出し，ディスクリプタに格納する．このため最初の実行時には，条件式の真偽にかかわらずその制御文全体を走査する．2回目以降は最初の実行で作成されたディスクリプタにより直接，制御構造中の特定の位置にジャンプする．このため，最初の実行と2回目以降の実行では，意味の解析や実行の手順とともに構文解析も切り換える．

（3） 実行の抑制

宣言として解析されている手続きの中では，制御文の実行や手続き呼び出しは行わず，式の評価は型の整合の検査だけを行って値の演算や代入は行わない．同様に，条件付きの制御文の最初の実行時には，条件式の値によって実行されない部分も構文と意味の解析は行う．さらに，プログラムを先頭から読んで直接実行するので前方に存在する手続きやラベルの位置は知られていないが，これは最初の参照時に，中間にあるプログラムを読み飛ばしながら探索する．これらはいずれも，構文と意味の解析は行いながら，実行は抑制

---

† Design and Evaluation of a Direct-Execution Algorithm on UDEC by KOZO ITANO (Institute of Information Sciences and Electronics, University of Tsukuba), YUTAKA SATO (Doctoral Program in Engineering, University of Tsukuba) and KENJI HAYASHI (College of Information Science, University of Tsukuba).
本研究は一部を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
††† 筑波大学工学研究科
†††† 筑波大学情報学類

する機能で実現する.

3. 実行文法の設計

構文の解析や意味の実行を，単純なハードウェアで高速に処理できるよう制約した文法をもとに，構文の各生成規則単位に意味を付加するために拡張を加えて，実行文法を設計した.

3.1 記述の形式

直接実行される言語はトークンごとに，かつ後戻りなしに構文を確定できる必要があり，ハードウェアで実現するのに十分に単純でなければならない．これを満足するものとして，生成規則の右辺の記号数を制限し，右辺の先頭にのみ必ず終端記号が現れるように構文を展開する Greibach の標準形がある[5),6)]．そこで基本的には右辺の非終端記号の数が2であるような Greibach 標準形を採用し，以下の点を拡張した.

（1） 非終端記号に対する選択肢が唯一であるときは選択が一意に行えるので，先頭の終端記号が空であることを許し，空の生成規則も許す.

（2） 終端記号の位置に意味解析や実行の結果を受け取って判断するための記号として TRUE と FALSE を書けるようにし，マッチングの対象としてトークンを受け取るか，意味の解析や実行の結果を受け取るか，あるいは空とするかの選択を，非終端記号ごとに明示的に書く.

（3） 各生成規則に対して，その生成規則が認識されたときに行うべき意味の解析や実行のための命令(意味命令)を対応付ける．各生成規則に持たせられる意味命令は実行の抑制と，字句解析部，意味解析実行部のそれぞれに対して最大一つとする.

3.2 直接実行アルゴリズムの記述例

図1および**図2**に，PASCAL の if 文を実行文法で記述した例を示す．図1は構文の認識を中心に字句解析，意味解析および実行の制御を記述し，構文意味認識部の制御テーブルに格納される．図2は意味の解析と実行を具体的に記述しており，マイクロプログラム化されて意味解析実行部の中に置かれる[1),2)].

（1） 構文認識と意味実行命令の生成の記述

図1の例で，右辺の最初に現れる〈t〉,〈r〉,〈 〉は，終端記号とマッチさせる対象として字句解析部からのトークンを使用するか，意味解析実行部からの実行結果を使用するかを指定する．%が前についた記号は，対応する生成規則の認識されたときに発せられる意味命令である．%K＋と%K－は実行の抑制を管理するスタックのプッシュ／ポップの指示，%L－ と %Lc は字句解析部へのトークンの差し戻しと先読みの取り消しの指示である．%Cn，%Dn（n は数字）は制御構造，データの処理に関係する意味の解析や実行の指示である．%En はマッチが失敗した場合に出力するエラー番号である.

（2） 意味解析と実行の記述

意味の解析と実行の具体的な記述は，図2の例のよ

```
<stmt>         ::= <t> { if      <if-1>                                %C7
                       | while   <while-1>                             %C14
                       | begin   <stmt-list> <comp-end>
                       | <id>    <proc-asg>                            %C6   }%E16
<if-1>         ::= <r> { TRUE    <b-expr> <fm-if>
                       | FALSE   <b-expr> <sm-if>                            }
<fm-if>        ::= <t> { then    <fm-if2>                                    }%E17
<fm-if2>       ::= <r> { TRUE    <stmt> <t-else>
                       | FALSE   <stmt> <f-else>           %K+               }
<t-else>       ::= <t> { else    <stmt> <t-else2>          %K+         %C8
                       | ;                                       %L-   %C9
                       | end                                     %L-   %C9   }%E18
<f-else>       ::= <t> { else    <stmt> <else-cls>         %K-         %C10
                       | ;                                 %K-   %L-   %C9
                       | end                               %K-   %L-   %C9   }%E18
<t-els2>       ::= < > {         <else-cls>                %K-               }
<else-cls>     ::= < > {                                               %C11  }
<sm-if>        ::= <t> { then    <sm-if2>                                    }%E17
<sm-if2>       ::= <r> { TRUE    <sm-if3>
                       | FALSE   <sm-if4>                        %Lc   %C12  }
<sm-if3>       ::= < > {         <stmt> <sm-if5>                             }
<sm-if4>       ::= <r> { TRUE
                       | FALSE   <stmt> <sm-if5>                             }
<sm-if5>       ::= < > {                                         %Lc   %C13  }
```

図 1 実行文法の記述例
Fig. 1 An example of the interpretation grammar.

```
%C7 :  push_s_n();
       search_cdt();
       if(found) {   get_cdt(-,-,-,-,&t); if(t==TRUE)       r=FALSE;
                                          else              r=TRUE; }
       else      {   put_cdt(IF,top_s_n(),-,-,-);           r=TRUE; }
%C8 :  search_cdt(); put_cdt(-,-,n_ptr,-,-);
%C9 :  search_cdt(); put_cdt(-,-,NIL,c_ptr,TRUE); pop_s_n();
%C10:  search_cdt(); put_cdt(-,-,n_ptr,-,-);
%C11:  search_cdt(); put_cdt(-,-,-,n_ptr,TRUE);   pop_s_n();
%C12:  search_cdt(); get_cdt(-,-,&t,-,-);
       if(t==NIL){   get_cdt(-,-,-,&n_ptr,-);     pop_s_n(); r=TRUE; }
       else      {   get_cdt(-,-,&n_ptr,-,-);                r=FALSE;}
%C13:  search_cdt(); get_cdt(-,-,-,&n_ptr,-);     pop_s_n();


c_ptr      : pointer to the first character of the current token;
n_ptr      : pointer to the next character to be fetched;
push_s_n   : pushes c_ptr onto s_n_stack;
pop_s_n    : pops s_n_stack;
top_s_n    : returns top of s_n_stack;
search_cdt : searches in CDT(Control Descriptor Table)
             for a descriptor whose key is the top of s_n_stack;
put_cdt    : stores values into the found descriptor;
get_cdt    : gets values from the found descriptor;
```

図 2 意味の記述例
Fig. 2 An example of descriptions for semantic actions.

うに基本的にはCに類似した構文で記述し，あらかじめ用意されているレジスタ群と，意味を処理するプリミティブ関数を使用する．この記述例の中で t は作業用のレジスタであり，found は関数 search_cdt によってセットされるフラグである．また，c_ptr は現在認識中のトークン，n_ptr は次のトークンの先頭文字をそれぞれ指すポインタ，r は意味解析実行部から構文意味認識部へ送る実行結果の値を設定するレジスタである．put_cdt または get_cdt は search_cdt で検索されたディスクリプタの各フィールドに値を設定または取り出す関数で，各引数がディスクリプタのフィールドに対応している．これらの関数の引数で‘-’は対応するフィールドへの作用がないことを指定する．

（3） 処理手順の例

図1の例では，構文意味認識部で〈stmt〉の展開時に if を認識したときに，意味解析実行部に対して意味命令%C7 を送るよう記述している．意味解析実行部での処理は図2の例のように記述され，ここではこの if 文のアドレスによって，この if 文のディスクリプタが登録されているかどうかを調べ，結果を構文意味認識部に返す．このとき，登録されていなければ新たにディスクリプタを作成して登録する．次の構文意味認識部における〈if-1〉の展開ではこの解析結果によって選択を行っている．最初の実行（ディスクリプタが登録されていないとき）では，条件式の評価の後〈fm-if〉が，２回目以降の実行のときは〈sm-if〉が指定される．このようにして，意味の解析や実行の結果を構文の解析にフィードバックしながら処理を進める．

3.3 意味解析手法の改良

図1および図2に示した実行文法の例は，すべての意味解析が構文の認識以降に実行されることを前提として設計されている．このため，意味解析の結果を構文解析に戻すために，パイプライン並列性が損なわれている．そこで，意味解析の一部を構文解析の前に移し，構文の認識の前に名前をその属性に従って分類して，構文解析と意味の実行の流れがスムーズになるように変更してみた．これを行うと，図1の〈stmt〉の生成規則のうち〈id〉が，変数か手続きか関数かなどに細分類され，後続の生成規則が一度にこのレベルで選択できる．ここで名前の属性を調べる際にスコープの処理も同時に行い，ブロック構造中での名前の同定はこの段階で済ませる．これは比較的に負荷の重い処理なので，後段にある意味解析実行部とのパイプラインのバランスの点でも有利な方式である．

4. 構文意味認識機構

実行文法の形で設計された直接実行アルゴリズムのうち，実際の意味実行手続きの実行部を除いた制御の中心となる部分をハードウェアで実行する構文意味認識機構 (SSR : Syntax and Semantics Recognizer) に

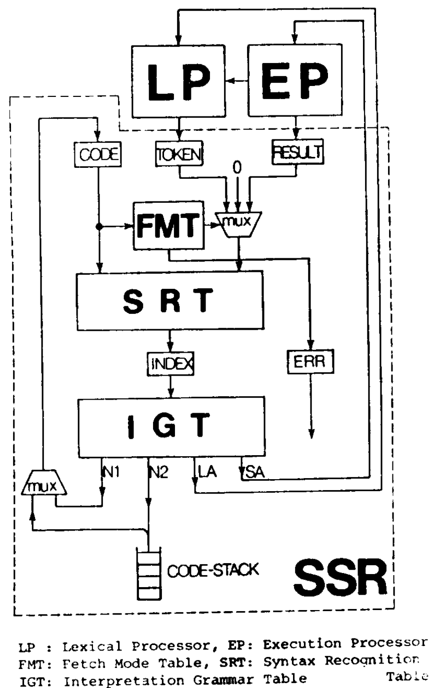


図 3 SSR のハードウェア構成
Fig. 3 Hardware organization of SSR.

ついて以下に説明する.

4.1 ハードウェアの基本構成

図 3 に SSR の基本構成を示す. SSR は制御テーブルとして FMT (Fetch Mode Table), SRT (Syntax Recognition Table), IGT (Interpretation Grammar Table) を含み, 現在までの構文解析状態を保持するスタックとして CODE-STACK, 現在の非終端記号を示すレジスタとして CODE, 現在の生成規則を示すレジスタとして INDEX を含む. このほか, 外部とのインタフェースのためのレジスタとして, 字句解析部からのトークンを保持する TOKEN, 意味解析実行部からの実行結果を保持する RESULT, エラーコードを保持する ERR がある.

制御テーブルのうち FMT は非終端記号ごとに, その展開時に右辺の終端記号にマッチさせる入力の選択指定 (〈t〉, 〈r〉, 〈 〉のうちの一つ) と, マッチするものがない場合のエラーコード %En を格納する. SRT は, 展開すべき非終端記号と FMT で選択される入力の組ごとに, 対応する IGT 中の生成規則のインデックスを格納する. 許されない組み合わせの場合には −1 を格納する. IGT は生成規則ごとに右辺の二つの非終端記号 N1, N2 および, 実行すべき意味 SA, LA などを格納する. SA は意味解析実行部への意味命令として%Cn または%Dn を格納する. LA は字句解析部を制御するための情報として %L−, %Lc を含む. これに加えて, SSR 自身を制御するための情報として%K+, %K−およびN1, N2 の存在の有無を示すビットなどがある. これらの制御テーブルを言語に応じて書き換えることで, 複数の言語に対応することができる.

4.2 動作原理

SSR は以下に示す 3 ステップを 1 サイクルとして一つの生成規則を認識する. そしてこれをプログラムの実行が終了するか, エラーを検出するか, または対話的実行のための割込みが発生するまで繰り返す.

Step1 : CODE の値をアドレスとして FMT から入力の選択指定を読み出し, それに従って TOKEN, RESULT, '0' のうちの一つをマルチプレクサで選択する. 同時に, エラーコードを ERR にセットする.

Step2 : CODE と Step1 で選択された値の組をアドレスとして SRT を読み出し, INDEX にセットする. このとき INDEX の値が −1 であるなら, ERR の値をエラーメッセージとして出力する.

Setp3 : INDEX をアドレスとして IGT のエントリを読み出し, N1 が存在するならそれを CODE にセットし, N2 が存在するならそれを CODE-STACK にプッシュする. 非終端記号が存在しないなら, CODE-STACK をポップしてそれを CODE にセットする. 同時に, SA が有効であるならそれを意味解析実行部に送り, さらに LA が有効であればそれを字句解析部に送る.

4.3 実　　現

試作した SSR のハードウェアは, TTL の IC を 47 個, NMOS のメモリを 10 個使用して実現された. 250 ns のクロックを用いているので, SSR 単体として 1 生成規則の認識に要する時間は 250 ns×3=750 ns である. 実行速度に関しては, 実際に他のモジュールと結合して動作させる場合は同期待ちになることが多くなるので, この程度で十分である.

5. 評　　価

直接実行の性能は, 1 トークンあたりの実行に要する時間を尺度として評価できる. 1 トークンあたりの実行時間は, SSR の性能と直接実行アルゴリズムの設計に依存するだけでなく, 字句解析部や意味解析実行部などの性能に関係する. これらを総合的に評価

表 1 実行文法で記述した直接実行アルゴリズムの静的解析
Table 1 Static analysis of the interpretation grammar representation of the direct execution algorithm.

	LR(1)	IG(A)	IG(B)	IG(C)
非終端記号数	26	98	92	88
生成規則数*				
ソーストークン		139	135	126
(名前の細分類)		—	—	33
解析・実行結果		24	27	18
空		26	22	28
合計	63	189	184	205
意味命令の種類		67	66	70
意味命令を持つ生成規則		86	111	121
意味命令を持つ空の生成規則		9	9	9

* 終端記号の種類による分類

するために，PASCAL のサブセット[4]を拡張した TINY-PASCAL を対象とする直接実行アルゴリズムを実行文法で記述して UDEC に実装し，この上で簡単なサンプルプログラムを実行し，SSR の動作を解析するとともに，直接実行アルゴリズムの設計の評価と最適化を行った．

5.1 実行文法による記述の規模

もとの PASCAL サブセットの構文は LR(1) で記述され，その規模は非終端記号数で 26，生成規則数で 63 であった[4]．これをもとに実行文法で記述した直接実行アルゴリズムでは，非終端記号の数が 88，生成規則の数は 205 となった．**表 1** に，実現された実行アルゴリズムにおいて各生成規則が持つ終端記号の種類の内訳，および意味命令数を示す．全体の中で，意味解析実行部からの実行結果を受け取るものは 9％であった．また，終端記号を持たない生成規則は 14％に抑えられており，そのうち 31％は非終端記号も持たない，意味命令を付加するために作られた空の生成規則である．機械的に強制的に展開すれば，このようなものを無くしてしまうことが可能であるが，それによって構文が，意味のつけ難い不自然な形に分割されてしまううえ，構文の規模が不必要に増大するので，現在以上に分割するのは得策ではないと考えている．

5.2 実行アルゴリズムの動的解析と改良

実行アルゴリズムの設計は，同一プログラムの実行のために行われる構文認識の回数で評価できる．この評価のために，サンプルプログラムとしてバブルソートプログラムを実行した結果の分析を**表 2** に示す．最初に設計されたアルゴリズム A では，このプログラムの実行のために 6,983 回の生成規則の認識が行われた．ここから，終端記号フィールドが空である生成規則を減少させるよう構文を組み替え，生成規則への意味の対応づけを改良するなどのアルゴリズム上の最適化を行って得られた B では，認識の回数が 3,917 に減少した．さらに，名前を構文解析の前に分類しておくという構成上の変更の上に実現されたアルゴリズム C では，これを 3,318 回まで減少できた．このような構文認識回数の減少とともに，実行された命令の総数は減少し，ディスクリプタテーブルへの参照回数はほぼ一定に抑えられているので，総合的な性能が改善されていることがわかる．

表 2 直接実行アルゴリズムの動作解析
Table 2 Dynamic analysis of the direct-execution algorithms.

	IG(A)	IG(B)	IG(C)
生成規則の認識回数*			
ソーストークン	3939	2389	1729
(名前の細分類)	—	—	653
解析・実行結果	1473	861	209
空	1571	667	727
合計	6983	3917	3318
意味解析実行命令数			
宣言の処理	28	28	22
制御の処理	588	568	410
データの処理	3208	2478	2003
合計	3824	3074	2435
ディスクリプタ参照数			
手続きディスクリプタ	205	205	660
制御構造ディスクリプタ	164	164	164
データディスクリプタ	1101	1238	700
合計	1470	1607	1524

のべ入力トークン数：1992

* 終端記号の種類による分類

5.3 SSR の性能

上述のように最適化されたアルゴリズム A では，表 2 に示すように，のべ入力トークン数 1,992 個に対して，生成規則の認識の総回数が 3,318 回，このうち字句解析部からトークンを入力したものが 2,382(1,729＋653) 回，意味解析実行部から実行結果を受けつけたものが 209 回であった．したがって入力された 1 トークンが平均して約 0.9 μs (1.2 回の SSR サイクル) で認識されている．また，UDEC 全体として 1 トークンを処理する際に，実行結果の受けつけも含めて SSR で費やされる時間は，平均して約 1.3 μs (1.7 回

の SSR サイクル) である．これに対して，字句解析部や意味解析実行部はこの数倍の実行時間を要するので[1),2)]，SSR の性能は現在の実現で十分であるといえる．

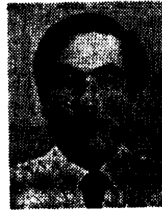
6. むすび

言語の構文と意味の認識および実行を一体的に記述する実行文法を導入し，この記述を高速に実行する専用のハードウェアとして構文意味認識機構 SSR を試作した．また実際に PASCAL のサブセットに対する直接実行型計算機を実装し，サンプルプログラムを実行して性能を評価し，SSR が十分高速に動作することを確認した．実行文法を導入して直接実行アルゴリズムの設計と最適化を比較的形式的に行うことによってその作業を軽減できたが，それでもなお意味解析や実行のアルゴリズムの設計は容易でなかった．この点を解決するには，意味を扱うためのプリミティブや，記述に関する戦略をさらに整理する必要がある．

参考文献

1) 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).
2) 板野肯三，佐藤 豊，杉原敏昭：マイクロプログラムで実現された直接実行型計算機のシミュレーションと評価，テクニカルノート HLLA-128，筑波大学電子情報工学系 (1986).
3) 林 謙治：ハードウェアによる構文意味認識機構の設計と実現，テクニカルノート HLLA-128，筑波大学電子情報工学系 (1986).
4) Aho, A. and Ullman, J.: *Principles of Compiler Design*, Addison-Wesley, Massachusetts (1979).
5) Greibach, S. A.: A New Normal Form Theorem for Context Free Phrase Structure Grammer, *J. ACM*, Vol. 12.
6) 房岡あきら：Algol Machine に関する一考察，情報処理，Vol. 11, No. 9, pp. 533-545 (1970).

(昭和 61 年 1 月 8 日受付)
(昭和 61 年 8 月 27 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技官，同大学電子情報工学系助手を経て，現在，同講師．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．現在同大学院博士課程工学研究科に在学中．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．

林 謙治（正会員）

昭和 39 年生．昭和 61 年筑波大学第三学群情報学類卒業．同年(株)日立製作所入社．在学中は高級言語計算機の研究に従事．コンピュータアーキテクチャに興味を持つ．