〈実践報告・資料〉

構文解析ルーチン作成支援ツールの開発

岐阜職業能力開発短期大学校　　宮　田　利　通

Implementation of computer aided parser generator

Toshimichi MIYATA

要約　短大の実技でコンパイラを制作するとき、学生が最も労力を必要とするのが構文解析ルーチンの作成作業である。これを省力化するためのツールを開発した。また、このツールは文法がLL(1)であるかどうか検査するためにも利用できる。短大の情報技術科の実技でこのツールを利用すれば、少ない実習時間を有効に使え、さらに技能レベルの低い学生でもコンパイラの制作が可能になる。制作したい言語の文法をツールに入力すると、C言語のプログラムとしての構文解析表を出力する。この構文解析表を使ったパーサーの部分はライブラリとして用意したので、ツールの利用者は字句解析ルーチン、コード生成ルーチン、その他関連ルーチンを作成すればよい。構文解析の方法は、再帰処理の無い予測的構文解析法を採用した。この方法は構文解析ルーチンが文法に依存しないという特徴を持ち、文法に対応した構文解析表とスタックを用いた効率の良い下降型構文解析法である。開発したツールは、5つの主要な部分から構成されている。文法から(1) FIRST 集合と(2) FOLLOW集合を計算し、これらの集合を使って(3) 構文解析表を作る。構文解析表は、コンパイラの開発に利用できるように(4) C言語のプログラムとして出力した。(5) 共通なプログラムはライブラリとして用意した。

I　はじめに

コンパイラを制作するとき、最も作成に手間の掛かる部分が構文解析ルーチンである。コンパイラを制作する教材（宮田　1992）が利用できるが、これは再帰下降解析による構文解析を実行している。再帰下降解析は、文法と対応が良い反面、文法が複雑になると構文解析ルーチンも複雑になり、作成が大変である。特に、構文解析ルーチンの作成が最も労力を必要とする。

上記の教材（実技教科書）を利用して短大生が、限られた実習時間内にコンパイラを完成するのはなかなか難しい。そこで、より短い時間でコンパイラを作成できるように、構文解析ルーチンの作成を助けるツールの必要性を強く感じた。構文解析ルーチン作成支援ツールは、LL(1)文法を入力して構文解析表を出力する。この場合、C言語でコンパイラを作成できるように、C言語プログラムとして出力した。この構文解析表を使った構文解析ルーチンの部分はライブラリとして用意したので、このツールの利用者は、字句解析ルーチン、コード生成ルーチン、その他関連ルーチンを作成すればよい。

UNIXで利用できる構文解析自動生成システムなどはLR(1)を採用している。しかし、前記したコンパイラの制作教材が再帰下降解析を使っているので、教育訓練上の関連を考慮してLL(1)を採用した。

なお、ツールの開発にはProlog言語を使ってプログラムを記述した。

II　ツールの概要

ツールで採用した構文解析の方法は、予測的構文解析法（Aho, Ullman　1977）である。予測的構文解析系は、スタック、構文解析表、制御プログラムから構成されている。図1に予測的構文解析系の構成を示す。

ツールは、翻訳文法（Pyster　1980）を入力してC言語で記述された構文解析表を出力する。スタックと制御プログラムは文法に依存しないので、ツールに付属したライブラリとして用意した。

予測的構文解析系の動作は、入力記号である終端記

論文受付日　1993.2.19

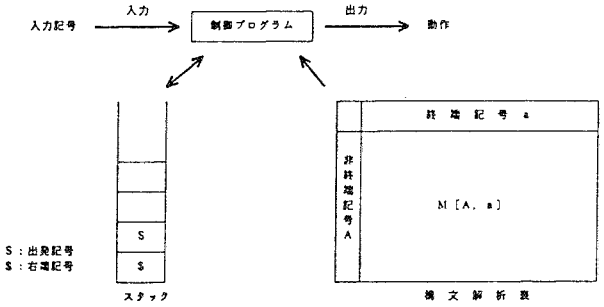
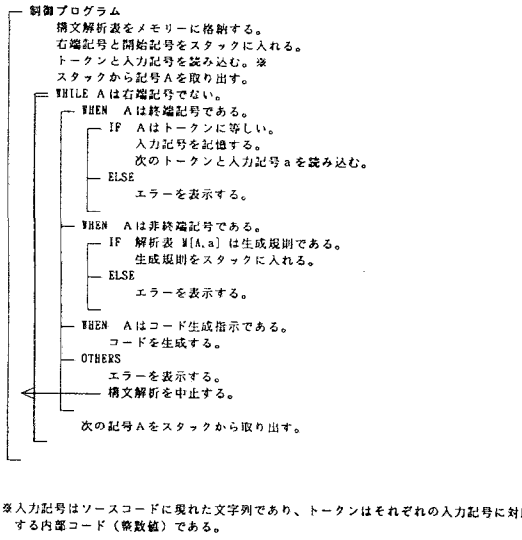


図1　予測的構文解析系の構成

号とスタックの最上段の記号を調べそれぞれの記号によって、次の入力記号を字句解析ルーチンから受け取る、構文解析表を参照しその内容をスタックへ格納する。あるいはコードを生成することである。図2に制御プログラムの行動ダイアグラムを示す。



※入力記号はソースコードに現れた文字列であり、トークンはそれぞれの入力記号に対応する内部コード（整数値）である。

図2　制御プログラムの行動ダイアグラム

制御プログラムの働きの概略は、次の通りである。

(1) 構文解析表をメモリー内に形成し、スタックを初期化する。

(2) スタックの内容が入力記号と等しければ、入力記号をコード生成のときに使用するため、一時記憶しておく。

(3) スタックの内容が非終端記号であれば、解析表の内容をスタックに格納する。

(4) スタックの内容がコード生成指示であれば、それに対応したコードを生成する。

コンパイラを制作するための教材で採用した再帰下降解析法と構文解析ルーチン作成支援ツールで採用した予測的構文解析法の特徴を比較してみる。

再帰下降解析法は、次のような特徴を持つ。

(1) コンパイラを記述するのに再帰処理の可能なプログラミング言語が必要である。

(2) 文法が複雑になると構文解析ルーチンも複雑になる。

(3) 文法を変更すると構文解析ルーチンも書き替えなければならない。

(4) 文法と構文解析ルーチンの対応がよいので理解しやすい。

一方、予測的構文解析法は、次のような特徴を持つ。

(1) 構文解析を非再帰的に実行できるので、どのようなプログラミング言語でも効率よく実現できる。

(2) 文法の複雑さに関係なく構文解析ルーチンが簡単である。

(3) 文法を変更しても構文解析ルーチンを書き替える必要がない。

(4) 複雑な文法の構文解析表を手作業で作成するのは難しい。

これら二種類の構文解析法を比較すると、予測的構文解析法は構文解析表の作成を自動化することで、予測的構文解析法の長所が生かされ、コンパイラの作成が容易になる。そこで、コンピュータに文法を入力するとC言語のプログラムとして構文解析表を出力するような機能を持ったツールを開発した。

III　ツールの構造

ツールは、四つの部分から構成されている。

(1) 入力された文法から各非終端記号のFIRST集合を計算する。

(2) 入力された文法から各非終端記号のFOLLOW集合を計算する。

(3) 文法、FIRST集合、FOLLOW集合から構文解析表を生成する。

(4) 構文解析表をC言語のプログラムに変換して出力する。

図3に構文解析ルーチン作成支援ツールのデータ・フロー・ダイアグラムを示す。

NとTをそれぞれ非終端記号と終端記号の集合としたとき、FIRST集合は、次のように定義される。

$$FIRST(A) = \{a \mid A \overset{*}{\rightarrow} au,\ a \in T,\ u \in (N \cup T)^*\}$$

すなわち、FIRST 集合はAから導出される終端記号列の長さ1の接頭語の集合である。ただし、$A \overset{*}{\rightarrow} \varepsilon$ ならば、εをFIRST (A) に含める。

FOLLOW集合は、次のように定義される。

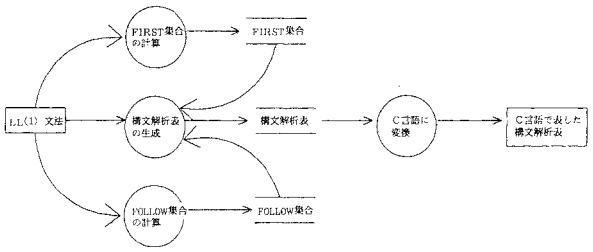


図3　構文解析ルーチン作成支援ツールのＤＦＤ

$$FOLLOW(A) = \{a \mid L \overset{*}{\rightarrow} uAv,\ a \in FIRST(v),\ u,v \in (N \cup T)^*\}$$

すなわち、FOLLOW集合は非終端記号Ａの右に現れる記号列の先頭の終端記号の集合である。ただし、Ａが出発記号ならば、右端記号＄をFOLLOW（A）に含める。

図４にFIRST 集合、図５にFOLLOW集合の計算の行動ダイアグラムを示す。

構文解析表の生成は、次のようにして求める。

(1)　文法の生成規則A→αのすべてに対して、以下のことを行う。

(2)　生成規則の右辺のFIRST 集合の元ａについて、A→αをM［A, a］に書き込む。

(3)　右辺のFIRST集合にεが含まれるならば、左辺のFOLLOW集合の元ｂについて、A→αをM[A, b］に書き込む。

(4)　構文解析表Mの未記入部分は、エラーとする。

すなわち、構文解析表は、現在の入力記号がａでありスタック・トップがＡであるとき、M［A, a］の部分に記入された生成規則でAを展開することを意味する。

図６に解析表を求める行動ダイアグラムを示す。

この構文解析表をコンパイラの作成に利用できるようにするために、Ｃ言語のプログラムに変換する。解析表は、次のようなデータ構造で表した。

```
struct M {
    int length,
        rule [p] ;
} M[q] [r] ;
```

ただし、p, q, r は、入力された文法からツール自身が計算して適正なサイズに決定される。

変換例として、文法の生成規則がS→uAvの場合を

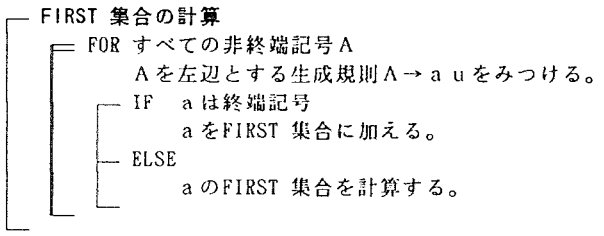
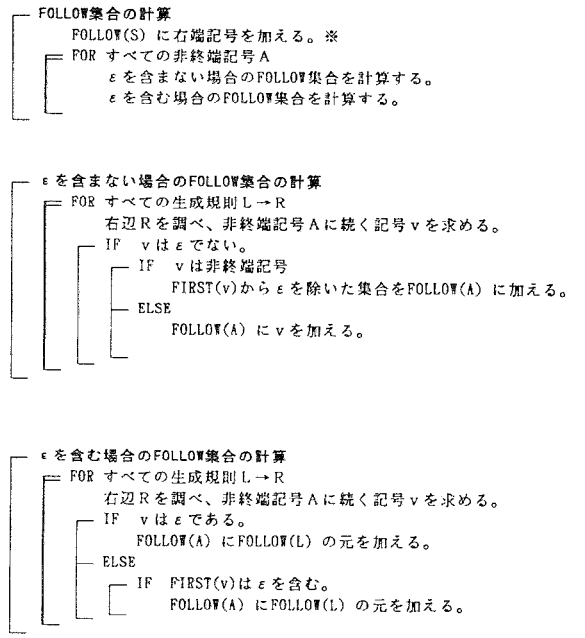


図４　FIRST集合の計算の行動ダイアグラム



※Ｓは出発記号である。

図５　FOLLOW集合の計算の行動ダイアグラム

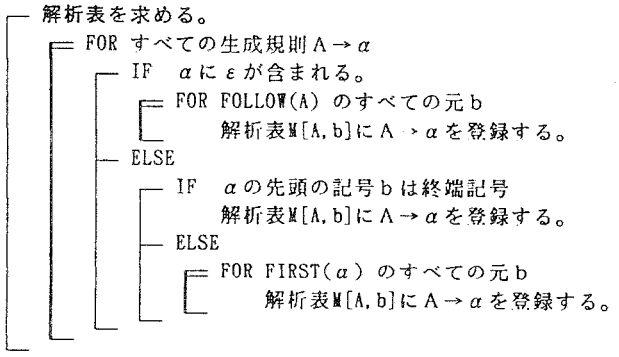


図６　構文解析表を求める行動ダイグラム

示す。なお、a∈FIRST(u)である。

```
M [low (S)] [col (a)]. length = 3;
M [low (S)] [col (a)]. rule [0] =u;
M [low (S)] [col (a)]. rule [1] =A;
M [low (S)] [col (a)]. rule [2] =v;
```

すなわち、lengthにはruleの要素数が代入される。関数lowと関数colは、Sとaの位置を計算して配列の添字としての整数値を返す関数である。S, a, u, A, vは、列挙定数として宣言されている。上記の構造体を要素とする二次元の配列は、関数make__LL__tableのC言語のプログラムとしてツールにより出力される。

次に、構文解析表の出力処理の内容を示す。

(1) マクロ定義文を出力する。
(2) エラーに関する列挙変数と列挙定数の宣言文を出力する。
(3) 終端記号に関する列挙変数と列挙定数の宣言文を出力する。
(4) 非終端記号に関する列挙変数と列挙定数の宣言文を出力する。
(5) コードに関する列挙変数と列挙定数の宣言文を出力する。
(6) 構文解析表の宣言文を出力する。
(7) 構文解析表をメモリー上に生成するために実行する関数を出力する。

構文解析表のプログラムは、systable.hというヘッダー・ファイルとして出力される。

構文解析ルーチンの共通部分は、parser.hというヘッダー・ファイルとして用意した。ファイルsystable.hは、parser.hの中から#include指令で読み込まれる。

IV　ツールの利用

コンパイラを制作するには、まず翻訳文法を作る。翻訳文法は、文法の中にコンパイルに必要な処理情報を記述したものである。この翻訳文法をPrologのデータベースとして取り込む。

具体例として代入文の場合を挙げる。図7に代入文の翻訳文法を示す。この文法で示される生成規則を図8で示すように記述する。すなわち、生成規則とそこで使われる記号の意味（出発記号、非終端記号、終端記号、コード生成指示、空列、右端記号）を記述する。

ツールを起動するとデータベースのファイル名を要求してくるので、図8で示された内容を格納したファイルの名前を入力する。ツールの実行が終了するとC言語のプログラムであるsystable.hが作られる。

構文解析ルーチンである制御プログラムはツールに付属したライブラリparser.hの中に記述されているので、ツールの利用者は、main関数、字句解析ルーチン、コード生成ルーチン、その他関連ルーチンを作成するだけで済む。なお、図7の翻訳文法の中に斜体文字で記述された実行コードと図8のコード生成指示とを対応付けるために、利用者は図9で示されるようなC言語プログラムを記述しなければならない。

図10にツールを利用したコンパイラ開発の手順を示す。

```
<assignment>  →  <id>x  push-operand(x)  =  push-operator(=)
                 <expression>  code(pop-operator, pop-operand, pop-operand)
                                                              {<id>}

<expression>  →  <factor>  <terms>                    { (, <id>, <integer> }
              |  +  <factor>  <terms>                             {+}
              |  -  <factor>  code(NEGATIVE, pop-operand)  <terms>
                                                                 {-}

<terms>       →  +  push-operator(+)  <factor>
                 code(pop-operator, pop-operand, pop-operand)  <terms>
                                                                 {+}
              |  -  push-operator(-)  <factor>
                 code(pop-operator, pop-operand, pop-operand)  <terms>
                                                                 {-}
              |  ε                                          {), ┤}

<factor>      →  <part>  <factors>                    { (, <id>, <integer> }
<part>        →  (  <expression>  )                                {(}
              |  <operand>x  push-operand(x)          {<id>, <integer> }
<factors>     →  *  push-operator(*)  <part>
                 code(pop-operator, pop-operand, pop-operand)  <factors>
                                                                 {*}
              |  /  push-operator(/)  <part>
                 code(pop-operator, pop-operand, pop-operand)  <factors>
                                                                 {/}
              |  ε                                       {+, -, ), ┤}

<operand>     →  <id>
              |  <integer>
```

図7　代入文の翻訳文法

```
/*  翻訳文法  */
$ASSIGNMENT$   ->  [$ID$, $PUSHOPERAND$, $EQUAL$, $PUSHOPERATOR$, $EXPRESSION$,
                    $CODEPOP$].
$EXPRESSION$   ->  [$FACTOR$, $TERMS$].
$EXPRESSION$   ->  [$PLUS$, $FACTOR$, $TERMS$].
$EXPRESSION$   ->  [$MINUS$, $FACTOR$, $CODENEG$, $TERMS$].
$TERMS$        ->  [$PLUS$, $PUSHOPERATOR$, $FACTOR$, $CODEPOP$, $TERMS$].
$TERMS$        ->  [$MINUS$, $PUSHOPERATOR$, $FACTOR$, $CODEPOP$, $TERMS$].
$TERMS$        ->  [$EPSILON$].
$FACTOR$       ->  [$PART$, $FACTORS$].
$PART$         ->  [$LEFTPAR$, $EXPRESSION$, $RIGHTPAR$].
$PART$         ->  [$OPERAND$, $PUSHOPERAND$].
$FACTORS$      ->  [$ASTERISK$, $PUSHOPERATOR$, $PART$, $CODEPOP$, $FACTORS$].
$FACTORS$      ->  [$SLASH$, $PUSHOPERATOR$, $PART$, $CODEPOP$, $FACTORS$].
$FACTORS$      ->  [$EPSILON$].
$OPERAND$      ->  [$ID$].
$OPERAND$      ->  [$INTEGER$].

/*  出発記号  */
sterm($ASSIGNMENT$).

/*  非終端記号  */
nterm($ASSIGNMENT$).
nterm($EXPRESSION$).
nterm($FACTOR$).
nterm($TERMS$).
nterm($PART$).
nterm($FACTORS$).
nterm($OPERAND$).

/*  終端記号  */
term($ID$).
term($INTEGER$).
term($EQUAL$).
term($PLUS$).
term($MINUS$).
term($LEFTPAR$).
term($RIGHTPAR$).
term($ASTERISK$).
term($SLASH$).
term($EPSILON$).
term($LF$).

/*  コード生成指示  */
cterm($PUSHOPERAND$).
cterm($PUSHOPERATOR$).
cterm($CODEPOP$).
cterm($CODENEG$).

/*  空列  */
eterm($EPSILON$).

/*  右端記号  */
dterm($LF$).
```

図８　翻訳文法のデータベース

```
void exccode(int stk_top, char *in_sym)
{
    switch (stk_top) {
    case CODEPOP:
        code(pop_operator(), pop_operand(), pop_operand());
        break;
    case CODENEG:
        code("NEGATIVE", pop_operand(), DUMMY);
        break;
    case PUSHOPERATOR:
        push_operator(in_sym);
        break;
    case PUSHOPERAND:
        push_operand(in_sym);
        break;
    case ERROR:
        printf("ERROR ¥n");
        break;
    default:
        break;
    }
}
```

図９　コード生成指示に対応付けたＣプログラム

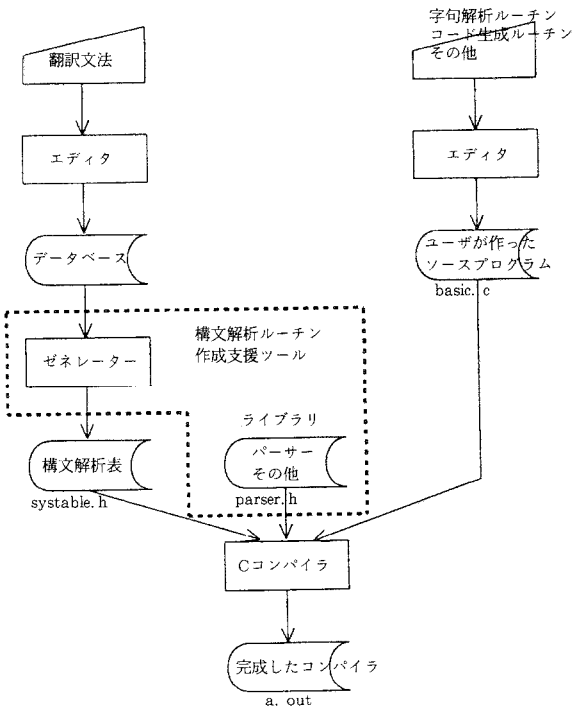


図10　コンパイラ開発の手順

V　むすび

コンパイラを制作する際、最もプログラム作成に労力を必要とする構文解析部分を簡単に作れるようになった。このツールを利用することにより、学生は翻訳文法を記述するだけで構文解析ルーチンのプログラミング作業を省力化できる。さらに、ツールの別の利用法として、文法がLL(1)であるかどうかを検査するためにも使える。コンパイラを制作するための実技教科書（宮田　1992）の中で記述されているBASICコンパイラのソース・コードと比較すると、このツールを使って作成したのならば、実技教科書のソース・コードの約75%を記述するだけで済む。

なお、構文解析作成支援ツールを職業能力開発に利用したい場合に限り、ソース・プログラムを無償で提供する。

謝　辞

この研究は、㈶小川科学技術財団から研究助成金を受けている。

参考文献

(1) A.V. Aho, J.D. Ullman:Principles of Compiler Design, Addison-Wesley, 1977

(2) B. Pyster: Compiler Design and Construction, Von Norstrand Reinhold, 1980

(3) 嵩、都倉、谷口：形式言語理論、電子情報通信学会、1988

(4) 宮田利通：コンパイラ制作のための教材開発、報文誌　第4巻第2号、職業訓練大学校、1992