[第5回目２００２・１０・1]

構文解析の基礎

これまで、式のインタプリターでは構文解析に topdown parser を使って解説した。top down parser は再帰的下方構文解析の代表的な手法であり、次に何が来るのかを推定しながら構文解析を進めていく方法である。比較的構成がわかりやすく、人手で書いていく場合などには適した方法とされている。
このほかにも構文解析には、上方構文解析法（bottomup parser、上昇型ともいう）という方法がある。この方法は人手で直接実現するには向かない方法であるが、理論的に構成されており、構文解析のプログラムを自動的に生成するためには重要な方法になっている。今回は、この下向き構文解析法についてみていく。

(1) E := E + T
(2) E:=T
(3) T := T * F
(4) T := F
(5) F := (E)
(6) F := id

簡単に説明するためにいつもの式の構文解析を考えてみる。文法は右のものを考える F,T,E は非終端記号であり、id は変数のようなシンボルを仮定する。

上向き構文解析と還元

構文解析の重要な役割は、入力がこの文法にあっているかどうかを認識することである。下方構文解析では、まず、Eであることを仮定して解析をはじめ、それぞれの非終端記号に対応する関数を呼び出し、最終的に必要な終端記号列になっているかを認識する方法であった。つまり、構文木という観点からみれば、構文木の根から葉に向かって解析を進めていく。（ここで、この文法は左再帰で書いてあるため、そのままでは top-down parser ができないことは前回説明したとおり）
これに対し、上方構文解析では葉すなわち終端記号から、根すなわち非終端記号へ向かって文法を適用して、最終的にEになっているかを認識する。例えば,a+c*d を考えてみる。これを token の列にしてみると、

id + id*id

さらに、(6)の文法を適用して、

F+F*F

(3)(4)の文法を適用して

F+T*F ⇒ T + T*F ⇒ T + T

さらに、(1)(2) を適用して

E + T ⇒ E

右文形式	handle	還元につかった規則
a + b * c	a	(6) (4) (2)
E + b * c	b	(6) (4)
E + T * c	c	(6)
E + T * F	T*F	(3)
E + T	E+T	(1)
E		

となって、認識される。この適用して、非終端記号に置き換えていくことを還元(reduction)と呼ぶ。上方構文解析で、構文木を構成する過程は、文字列を非終端記号に還元していく過程である。この例では、順番を考えずにできるところから還元していったが、これをするためには入力を全部みてからやることになるため、あまり現実的ではない。
上向き構文解析では、入力を左から右に見ながら（つまり、一文字づつ入力しながら）還元していく。入力の右側から適用できる構文規則を逆順にたどって最終的に最後の構文規則まで還元できる部分列をhandle(把手)という。上方構文解析はhandleを見つけて還元する過程とみなすことができる（上図）。
このような構文解析を（自動的に）構成するために、現在の構文解析の状態を記憶するためのスタックと入力の動作として以下のものを考える。

1．移動(shift)：次の入力記号をスタックの上段に移動する。
2．還元(reduce)：handle の右の記号がスタックの一番上にあり、適用できる構文規則をみつけて、その非終端記号に置き換える。
3．受理(accept)：構文解析が終了
4．エラー：適用できる構文規則がみつからず、誤りを発見。

これを図示すると右のようになる。

スタックの状態	入力	動作
$	a + b * c $	shift
$a	+ b * c $	(6)(4)(2)による reduce
$E	+ b * c $	shift
$E +	b * c $	shift
$E + b	* c $	(6)(4)による reduce
$E + T	* c $	shift
$E + T*	c $	shift
$E + T*c	$	(6)による reduce
$E + T*F	$	(3)による reduce
$E + T	$	(1)による reduce
$E	$	accept

演算子順位構文解析法

さて、上方構文解析のためのアルゴリズムについて、考えていくことにしよう。演算子順位構文解析法は、演算子順位文法(operator precedence grammar)に対する簡単な上方構文解析法である。演算子文

法とは、どの構文生成規則の右辺も空ではなく、しかも、隣接する２つの非終端記号を持たないという文法である。つまり、算術式 E := E + T のように、必ず非終端記号の間には演算子（終端記号）が入るものである。演算子順位文法とは、終端記号について、優先度> < =が定義されている文法のことである。

１．A:= ...st...または、A:= ...sBt..なる構文規則があれば、s=t

２．A:= ...sB...なる構文規則があり、さらに B⇒t または B⇒Ct なる規則が導かれるならば、s < t

３．A:= ...Bt...なる構文規則があり、さらに B⇒...s または B⇒...sC なる規則が導かれるならば、s > t

例についていえば括弧(,)に関しては=、E:= E**+T** と T:=T*F により、+<*である。つまり、数式の直感的な優先度に対応している文法とおもえばよい。２，３の規則は構文規則で、構文木を作ったとき下流にある演算子が強いことを示す。このような関係にしたがって、構文規則より、演算子順位行列を作ることができる。

これを使って、以下のアルゴリズムをつかえばよい。

１．スタックに空記号$をつんでおく

２．入力記号 a をよむ

３．スタック上の演算子 s に対し、s > a であるかぎり、還元する

４．そうでなければ、a をスタック上につみ、２へ

５．全部認識されたら終わり。

	+	*	(	)	id
+	>	<	<	>	<
*	>	>	<	>	<
(	<	<	<	=	<
)	>	>		>	
id	>	>		>	

最後の reduction のために、便宜的に優先度が一番低い最後の記号を導入する必要がある。また、１項演算子を扱う場合には工夫が必要となる。

ＬＲ構文解析法

演算子文法に関しては比較的簡単なアルゴリズムで構成することができたが、一般の文法には使えない。ここで説明する方法は入力を左から右へ走査し、最右の規則を導くので、LR(left-to-right scanning right most derivation)構文解析法と呼ばれるものである。

LR 構文解析は入力とスタック、構文解析表からなる。入力は１記号ずつ左から右に読む。スタックには、

$s_0\ X_1\ s_1\ X_2\ s_2\ X_3\ s_3 \ldots .\ X_m\ s_m$

という記号列を積む。ｓは状態に対応した記号である。Ｘは文法記号で、実際は必要ないが説明のために加えてある。構文解析表は構文解析動作関数 ACTION と行き先関数 GOTO の２つの部分からなる。LR 構文解析のプログラムは現在のスタックの最上段の状態 s_m と入力記号 a_i をもちいて、ACTION[s_m, a_i]を引いて、以下の動作のどれかをとる。

（１） shift s: 入力記号 a_i と GOTO[s_m,a_i]で求めた次の状態 s をスタックに積む。次の入力に進む。

（２） reduce A := b: 文法規則 A:=b で還元する。すなわち、最上段にある X の列が b であるはずなので、これに対応する Xs のペアをスタックから取り除き、最後の状態 s_m と A で、GOTO[s_m,A]=s を次の状態とし、As をスタックに積む。還元の動作は現在の入力記号は変わらない。

（３） accept

（４） error

例に挙げた数式の文法について番号をつける。構文解析表は右のようになる。ここで、s*i* は shift で状態 *i* をスタックに積む動作を意味する。また、r*j* は文法 *j* による reduce 動作を意味する。ここで、a*b+c$を入力として考えてみよ。

(1) まずはじめの状態は０から始まる。

(2) state 0 で、id が入力されると s5 となっているので、shift。入力記号 id と状態５をつむ。

(3) 次に*が入力になるが、state 5 で、* の欄は、r6 である。これは文法（６）による reduce 操作である。スタックの上

	ACTION						GOTO (非終端記号)		
state	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

にある id 5 のペアを取り除き、最上段が 0 になるので、state0 と F を GOTO で引き、3 となっているため、F と 3 がスタックに積まれる。

(4) 入力記号はそのままである。state3 において、入力が* であれば、r4 である。文法規則(4)での reduce をする。T となり、state0 と T で GOTO を引き 2 となるため、T2 を積む。

(5) state 2 で、入力が*の時には s7 となる。したがって、* と 7 をつみ、次の入力に移る。

(6) 以下、省略。

このような表を作ることによって、LR構文解析ができる。この表の作りかたについてはこの講義では説明しないが、重要な点は字句解析のところでオートマトンから字句解析を自動的に生成できると同様に、この表を自動的に作る方法があり、構文解析ルーチンを自動的に生成できることである。

	スタック	入力
(1)	0	id*id+id $
(2)	0 id 5	*id+id$
(3)	0 F 3	*id+id$
(4)	0 T 2	*id+id$
(5)	0 T 2 * 7	id+id$
(6)	0 T 2 * 7 id 5	+id$
(7)	0 T 2 * 7 id F 10	+id$
(8)	0 T 2	+id$
(9)	0 E 1	+id$
(10)	0 E 1 + 6	id$
(11)	0 E 1 + 6 id 5	$
(12)	0 E 1 + 6 F 3	$
(13)	0 E 1 + 6 T 9	$
(14)	0 E 1	$

構文解析生成プログラム yacc

LR 構文解析ルーチンを自動生成するプログラムの一つが yacc である。実際、構文解析ルーチンは top-down parser で書くことがあるが、複雑になると手に負えなくなるため、yacc のような自動生成プログラムを使う。（linux のフリーな構文解析は実際 bison というプログラムであるが、yacc というコマンドになっている）実際の場面では yacc の使い方を習得しておくことが重要になる。

yacc は、LR 構文解析に一文字の先読み機能を付け加えた LALR(Look-ahead LR)という文法のクラスを扱うことができる。yacc の入力（文法の定義）は、例として以下のように書く。

```
%token NUM  /* yylex から返す token の定義、文字を直接返してもよい*/
%token SYM
%token STRING
%{
#include <stdio.h>  /*Cのプログラムのヘッダー、なんでもかける*/
%}
%start expression /* yyparse で何の認識をするかの指定*/
%% /* 文法の定義の始まり*/
expression: term
       | expression '+' term
       ;
term: factor
       | term '*' factor
       ;
factor: NUM | SYM ;
%% /* 文法の定義の終わり*/
#include "lex.c" /* ここからは何のCのプログラムをかいてもいい*/
```

token で定義されているものは、define されるので、lex.c のなかでそのまま使うことができる。lex.c では、これまででやった字句解析のルーチンが定義する。構文解析から呼び出される字句解析のルーチンは、yylex という名前で定義しなくてはならない。これは、lex を使っても生成できる。

このプログラムを expression.y とすると、

```
  % yacc expression.y
```

で、構文解析プログラムが y.tab.c という名前で生成される。このプログラムには、構文解析ルーチン yyparse が含まれており、、main プログラムを以下のようにして、リンクすればよい。

```
main(){
     yyparse();
     printf("ok¥n");
}
void yyerror(){  printf("syntax error!¥n"); exit(1); }
```

yyerror は構文解析でエラーになったときに呼び出される関数で、ユーザが与える。

簡単なC言語：tiny C

Lisp では、プログラムは括弧の式で書き、いわば中間表現をそのまま人間が手で書いているようなものであった。これはこれで、Lisp 言語ではプログラムをリストのデータ構造として扱えたりして、重要な機能の一部になっているが、やはり、書きにくいし、なれないと読みにくいなどの欠点がある。C や Java などほとんどのプログラミング言語ではその表記は読みやすく工夫されている。

これからは、Cのサブセットのような文法を持つ言語 tiny C を作っていくことにする。これまでつくった Lisp インタプリターを使って tiny C のインタプリターをつくる。この後、この言語のコンパイラを作ることを考える。

では、tiny C の文法を考えてみることにする。BNF 記法で右のように定義してみる。

```
<program> := { <function-definition> }*
<function-definition> :=
        <function-name> "(" <parameter-list> ")"   <function-body>
<parameter-list> :=  <> | <variable> { "," <variable> }*
<function-body> :=  "{"   <local-variable-decl>  { <statement> }* "}"
<local-variable-decl> := <> |  "var"   <local-variable-list> ";"
<local-variable-list> := <variable> { "," <variable> }*
<statement> := <assignment-statement> | <function-call-statement> |
            <if-statement> | <return-statement>
<assignment-statement> := <varaible> "=" <expression> ";"
<function-call-statement> := <function-name> "(" argument-list ")" ";"
<argument-list> := <> | <expression> { "," <expression> }*
<if-statement> := "if" "(" <expression> ")" <statement> "else" <statement>
<return-statement> := "return" <expression> ";"
<expression> := <expression> <term-op> <term>
<term-op> := "+" | "-" | "<" | ">"
<term> := <variable> | NUMBER | STRING | <function-call-expression>
<function-call-expression> := <fucntion-name> "(" <argument-list> ")"
<function-name> := SYMBOL
<variable> := SYMBOL
```

<expression>のあたりではだいぶ省略してある。この言語の仕様に従えば、例えば次のようなプログラムを書くことができる。

```
main(){
  println("foo  is %d",foo(10,1); )
foo(x,y){ var t;  if(x > y) t = x + y; else t = x - y; return t; }
```

このプログラムは、Lisp では、

```
(define main () (println "foo is %d" (foo 10 1)))
(define foo (x y)
   (block (t) (if (> x y) (= t (+ x y)) (= t (- x y)) (return t)))
```

つまり、構文解析により、上のプログラムを入力し、下の Lisp の内部データ構造を使えば、そのままインタプリターで実行すればよいことになる。

この構文解析を前に紹介した top-down parser で書くと、だいぶ複雑になる。そのため、通常、このような parser は yacc などのツールを使って作られるのが普通である。次に、yacc を使って tiny C のインタプリターを作ってみる。

演習課題５：

上記の yacc のプログラムを拡張し、演習問題１でやった括弧をいれた式を構文解析する yacc のプログラムを作りなさい。コンパイル、実行し、(a + b)*c + 1 が認識できることを確かめなさい。