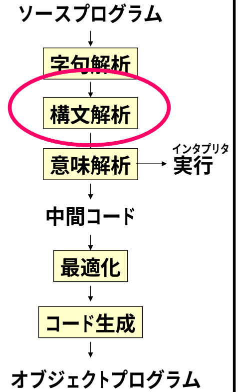
第３回（平成１５年度９月１６日）

構文解析の基礎と実際
yaccの使い方（１）

筑波大学　佐藤

言語処理系の基本構成

- 字句解析(lexical analysis): 文字列を言語の要素（トークン、token）の列に分解する。
- 構文解析(syntax analysis): token列を意味を反映した構造に変換。この構造は、しばしば、木構造で表現されるので、抽象構文木（abstract syntax tree）と呼ばれる。ここまでの言語を認識する部分を言語のparserと呼ぶ。
- 意味解析(semantics analysis): 構文木の意味を解析する。インタプリターでは、ここで意味を解析し、それに対応した動作を行う。コンパイラでは、この段階で内部的なコード、中間コードに変換する。



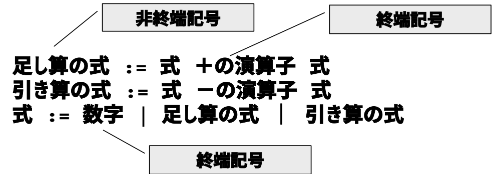
---

top-down parser

- これから、読み込むべきものを仮定して、それに合致するかを調べていく構文解析法
  - 構文規則の上位から下位に向かって、parserを構成する。
- 構文規則から、直感的なプログラムを構成する。
- 人間にわかりやすい。人手でparserを書く場合に有効な方法。

構文規則のおさらい

- BNFによる数式の構文規則
- 構文規則の最終的な要素＝終端記号（terminal)
- ほかの構文規則によって定義される記号
  ＝　非終端記号(non-terminal)



構文規則の定義

- 定義
  1. 構文規則は，非終端記号<T>に対して、<T> := e （eは構文規則）で、表現される。これは、非終端記号<T>は、構文規則eによって置き換えられることを意味する。
  2. eは空でもよい。
  3. eは、非終端記号、終端記号、もしくはe1 | e2、e1 e2、e1*のいずれか。e1 | e2は、e1もしくはe2であることを意味し、e1 e2 はe1の次にe2が現れることを意味し、e1*は、e1の0個以上の繰り返しを意味する。

- (...)は、構文規則のまとまりを示す
- e1|e2|e3は、((e1|e2)|e3)　と同じ
- e1 e2 e3　は（(e1 e2) e3）と同じである。
- 正規表現と似ている

---

tiny Cの構文規則

- BNF記法による

```
program :=  {external_definition}*

external_definition:=
           function_name '(' [ parameter {',' parameter}* ] ')' compound_statement
          | VAR variable_name ['=' expr] ';'
          | VAR array_name '[' expr ']' ';'

compound_statement:=
        '{' {local_variable_declaration}* {statement}* '}'

local_variable_declaration: =
          VAR variable_name [ {',' variable_name}* ] ';'

statement :=
           expr ';'
          | compound_statement
          | IF '(' expr ')' statement [ ELSE statement ]
          | RETURN [expr ] ';'
          | WHILE '(' expr ')' statement
          | FOR '(' expr ';' expr ';' expr ')' statement

expr:=    primary_expr
```

tiny Cの構文規則

- BNF記法による

```
expr:=    primary_expr
        | variable_name '=' expr
        | array_name '[' expr ']' '=' expr
        | expr '+' expr
        | expr '-' expr
        | expr '*' expr
        | expr '<' expr
        | expr '>' expr

primary_expr:=
          variable_name
        | NUMBER
        | STRING
        | array_name '[' expr ']'
        | function_name '(' expr [{',' expr}*] ')'
        | PRINTLN  '(' STRING ',' expr ')'
```


top-down parserとbottom up parser

- 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。
- *上方構文解析法*（*bottom-up 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。

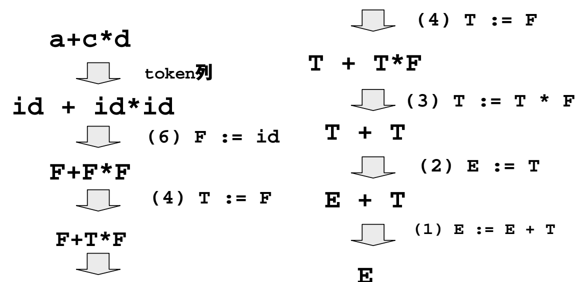
top-down parserとbottom up parser

- 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
- 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
- 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

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


bottom-up parserの例

- a+c*d　を考える



還元(reduction)

- 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
- 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

handleをつかった還元

- 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

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	---	---

bottom-up parserのアルゴリズムの構成

- bottom-up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。
  - 移動(shift)：次の入力記号をスタックの上段に移動する。
  - 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。
  - 受理(accept)：構文解析が終了
  - エラー：適用できる構文規則がみつからず、誤りを発見。

例

- 移動(shift)：次の入力記号をスタックの上段に移動する。
- 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。
- 受理(accept)：構文解析が終了
- エラー：適用できる構文規則がみつからず、誤りを発見。

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

- 演算子順位文法(operator precedence grammar)に対する簡単な上方構文解析法
- 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
  - 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
- 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
  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 かられるならば、$s < t$
  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、$s > t$

演算子順位構文解析法

- 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
- アルゴリズム
  - スタックに空記号$をつんでおく
  - 入力記号aをよむ
  - 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
  - そうでなければ、aをスタック上につみ、２へ
  - 全部認識されたら終わり。
- 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

演算子順位行列

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

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
- *LR(left-to-right scanning right most derivation)構文解析法*
  - 入力を左から右へ走査し、最 右の規則を導く
- 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 s0 X1 s1 X2 s2 X3 s3.... Xm sm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

- 構文解析動作関数ACTION
- 行き先関数GOTO
- 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smと入力記号aiをもちいて、ACTION[sm, ai]を引いて、 動作する

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			

LR構文解析のアルゴリズム

- 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smと入力記号aiをもちいて、ACTION[sm, ai]を引いて、 以下の動作のどれかをとる。
  - shift s: 入力記号aiとGOTO[sm,ai]で求めた次の状態sをスタックに積む。 次の入力に進む。
  - 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smとAで、GOTO[sm,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。
  - accept
  - error
- siはshiftで状態iをスタックに積む動 作を意味する。
- また、rjは文法jによるreduce動作を意味する

state	ACTION id	+	*	(	)	$	GOTO 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			

	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	$

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、state 0とTでGOTOを引き2とな るため、T 2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。

まとめ

- bottom up paraserがアルゴリズム的に構成できること。
- shiftとreduceの意味について覚えておくこと
- どうやって、構文解析表を構成するかについては、興味のある人は自分でしらべてみること。

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
- 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の書き方

- 数式の例

```
%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  /*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*/
```


yaccのつかい方

- 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

yaccのつかい方

- コマンド
  - プログラムをexpression.y とすると
    - % yacc expression.y
- 構文解析プログラムがy.tab.cという名前で生成される
- mainプログラムを以下の ようにして、リンクすればよい
- 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。

```
main()
{
      yyparse();
      printf("ok¥n");
}

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


yaccの動作

- yaccの動きは、以下のように動作する
  1. yylexを呼び出して、tokenを読み込み、そのtokenから始まる文法規則を探す。
  2. その文法規則が終るまで、tokenを読み、遷移(shift)を続ける。
  3. 文法に非終端記号がある場合は、その文法規則をスタックに積み、1からやり直す。
  4. 文法規則の最後まで遷移したら、その規則を還元(reduce)する。
  5. スタックから一つ前に処理していた規則に戻り、3.で還元した非終端記号をつかって、さらにshiftする。
  6. 2に戻る。

yaccのactionと意味値(semantic value)

- 構文解析の仕事は定義した構文にあっているかとチェックするとともに、構文を表現する構文木(抽象構文木： abstract syntax tree, AST)を作ることである
  - yaccのプログラムでは、単に構文が定義した文法にあっているかをチェックするものであった
  - 構文木は意味解析でその意味に従った処理が行われる。

yaccのaction

- yaccでは、構文解析の途中で、何らかの動作を行うactionの指定ができる。
- 文木を作る作業はこのactionの中で行う。actionは構文規則の中に{ } で囲んで, C言語で記述する。
- termの各規則がreduceされたときに、{}の中のactionが実行される。
  - 通常、actionは各構文規則の最後に書き、reduceされた時に実行されるようにするが、途中に書いてもよい。その場合には、そこまで、shiftされたときにactionが実行されるようになる

```
term:   factor { printf("factor is coming"); }
      | term '*' factor { printf("factor is addec
      ;
```

term * factorが認識されたら、実行される

意味値(semantic value)

- yaccでは各構文規則で生成される値を意味値(semantic value)を持つことができ、その構文で認識された構文木を意味値として、actionでその意味値を生成（計算する

```
term: factor { $$ = $1; }
    | term '*' factor
              { $$ = makeAST(PLUS_OP,$1,$3n); }
    ;
```

factorの意味値

結果として認識されたtermの意味値

termの意味値

意味値のデータ型の定義

- %unionは意味値のデータ型を定義する
- tokenはこれをつかって定義

```
%union {
  AST *val;
}

%type<val>   term factor
```

ASTのデータ型を持つ意味値はvalという型を持つ

termとfactorのtokenは、valすなわちAST *というデータ型と宣言する

終端記号の意味値

- factorでは、終端記号の意味値を使う
- 終端記号に対しては、字句解析ルーチンyylexからは、yylexの値をNUM,SYMを返すとともに、意味値をyylvalという変数を介して、意味値を返す
  - yylval.valで参照する。

```
%type <val> NUM SYM
 ...
factor: NUM  | SYM ;


int yylex(){
   .... /* NUMの時 */
   yylval.val = makeNum(n);
   return NUM;
  .... /* SYMの時*/
  yylval.val =
        makeSymbol(yytext);
  return SYM;
  ....
}
```


次回は

- tiny Cの構文解析
- 優先度の定義
- あいまいな文法とshift/reduce conflict, reduce/reduce conflict
- エラー回復処理

課題３

- 変数とプリント関数を持つ式を計算する言語の構文解析プログラムをyaccを用いて作成しなさい。

```
<program> := {<statement> ';'}*
<statement> := <assignment> | <print_statment>
<assignment> := <variable> '=' <expression>
<print_statemnt> := 'print' <expression>
<expression> := <expression> <op> <expression> |
                <variable> |
                <nubmer> |
                '(' <expression>')'
<variable> := {英字}*
<nubmer> := {数字}*
<op> := '+' | '-' | '*' | '/'
```

<variable>は、アルファベットからなるシンボルで、
<nubmer>は数字の並びで、
各tokenはcと同様に空白で区切られているものとする。
演算子の優先度を考慮すること。

- これは、Cのmainのみの機能がある言語である。例えば、以下のようなプログラムをかくことができる。

```
x = 1+2;
y = 100;
z = (x+y)*10+34;
print z+1;
```

- このプログラムを入力し、認識できることを確かめなさい。
- なお、構文木は必ずしもつくらなくてもよい。