プログラミング言語処理

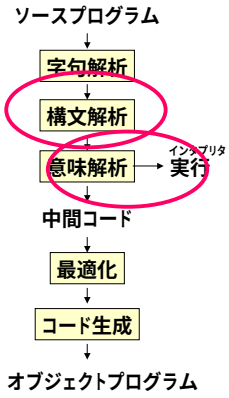
第４回（平成１５年度９月３０日）

構文解析の実際
yaccの使い方（２）
tiny Cの構文解析

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

```
a+c*d
   token列
id + id*id
   (6) F := id
F+F*F
   (4) T := F
F+T*F
```

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

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

プログラミング言語処理

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。

プログラミング言語処理

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

プログラミング言語処理

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

プログラミング言語処理

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			

プログラミング言語処理

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

プログラミング言語処理

	ACTION						GOTO		
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			

	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動作である。スタックの上にあるid 5のペアを取り除き、最上段 がo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でお 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Tをつみ、次の入力に移る。
6. 以下、省略。

プログラミング言語処理

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			

	スタック	入力
(1)	0	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5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、*の個は、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T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でる 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T7をつみ、次の入力に移る。
6. 以下、省略。

プログラミング言語処理

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			

	スタック	入力
(1)	0	id*id $
(2)	0	*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5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、*の個は、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T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でる 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T7をつみ、次の入力に移る。
6. 以下、省略。

プログラミング言語処理

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			

	スタック	入力
(1)	0	id*id $
(2)	0 id 5	*id+id$
(3)	0	*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5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、*の個は、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T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でる 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T7をつみ、次の入力に移る。
6. 以下、省略。

プログラミング言語処理

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			

	スタック	入力
(1)	0	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5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、*の個は、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T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でる 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T7をつみ、次の入力に移る。**
6. 以下、省略。

プログラミング言語処理

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			

	スタック	入力
(1)	0	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5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、*の個は、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T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でる 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T7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。（実際は、複数ある）
  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。

プログラミング言語処理

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。

プログラミング言語処理

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 adde
     ;
```

term * factorが認識されたら、実行される

プログラミング言語処理

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(MUL_OP,$1,$3); }
     ;
```

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

プログラミング言語処理

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

プログラミング言語処理

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;
  ....
}
```

プログラミング言語処理

優先度の定義

- yaccはLALR parserであり、一文字先読みをしているため、演算子の結合規則と、優先度を定義できる
- %leftは左結合規則を持つ演算子であることを指定する

```
%left '+'

expr: expr '+' expr { ...} ;
```

x + y + z に対して((x + y) + z)のように処理される

- %rightは右結合規則を持つもので、例えば代入の'='は右結合規則を持つものである
- 優先度は、結合規則が後に定義されたものが高いことになる。

実際の例

- 数式の優先度

```
%left '+' '-'
%left '*'
%left UMINUS
...
expr:   factor
       |  expr '+' expr { $$=addSymbol(plusSym,makeList2($1,$3)); }
       | exp '-' exp { $$=addSymbol(minusSym,makeList2($1,$3)); }
       | exp '*' exp { $$=addSymbol(mulSym,makeList2($1,$3)); }
       | '-' exp %prec UMINUS { .... }
    ;
```

この順番で、優先度が定義される

%precは単項演算子を最も優先度の高い処理をするための指定

あいまいな文法

- 文法にあいまいさがあると、LR 構文解析ができなくなるので、yacc は警告メッセージをだす。
  - ２種類あり、 *shift/reduce conflict, reduce/reduce conflict*がある。
- shift/reduce conflictとは、文法規則が shift（つまり、さらに長い非終端記号にreduceできる）なのか、reduce(そこで打ち切って、非終端記号にしてしまう)か、解釈ができることを示す。
  - この conflictは一概に文法定義が間違っているということではない場合がある。有名な例として、IF文の定義がある。

```
statement : IF '(' expression ')' statement
          | IF '(' expression ')' statement ELSE statement
           ....;
```


IF文の例

- 一般に yacc は、shift/reduce conflict がおきたときには、例外条件として、遷移(shift)を優先させる。したがって上の else は内側の if 文の一部と解釈される。この解釈は、C 言語を始めほとんどの言語の仕様と一致するので、一般に if 文にまつわる shift/reduce conflict はそのままにしておいて問題ない

```
if (a > 0)
  if (b > 0) c = 100;
   else
  c = 2000;
```

```
if (a > 0)
  if (b > 0) c = 100;
   else
  c = 2000;
```

どっち？

```
statement : IF '(' expression ')' statement
          | IF '(' expression ')' statement ELSE statement
           ....;
```


reduce/reduce conflict

- reduce/reduce conflictは、同時に還元できる文法規則が複数あることを意味する
- 便宜上、yaccでははじめに現れた文法規則を優先させるが、これは望ましいことではないので、このconflictがないように文法を作る必要がある

例

- 0個以上のword列を読む場合

```
sequence: /*  */ { printf ("empty sequence¥n"); }
        | maybeword
        | sequence word { printf ("added word %s¥n", $2); }
        ;
maybeword: /*  */ { printf ("empty maybeword¥n"); }
        | word { printf ("single word %s¥n", $1); }
        ;
```

wordはmaywordにreduceでき,sequenceでもreduceできてしまう

```
sequence: /*  */ { printf ("empty sequence¥n"); }
        | sequence word { printf ("added word %s¥n", $2); }
        ;
```


左再帰

- 再帰する場合、yaccでは、right recursionでは、途中の状態をスタックにとっておく必要があるため、なるべく、left recursionで書いておくべき

```
seq: item | seq ',' term ; /* left recursion */
```

こっちのほうが望ましい

```
seq: item | term ',' seq ; /* right recursion */
```

エラー回復処理

- 通常使っているコンパイラでは、途中で文法エラーを見つけたとしてもなるべく、他の部分もparseして一度に多くの文法エラーを見つけることができるようにしてある。
- 文法エラーを見つけたときに、次にどこから構文解析を再開するかの処理をエラーからの回復処理という。どこから処理を再開するか、どうやって再開するかについてはコンパイラの使いやすさの要素の一つにもなり、結構むずかしい問題である。

yaccのエラー回復処理

- yaccでは、予約の非終端記号として、errorという予約語があり、yyerrorが呼び出されて、これが終了(retrun)すると、errorという記号にreduceされるように処理してある。

```
statement: ....
         | error ';'
         ;
```

statementの構文解析で文法エラーが起きた場合には、';' がくるまで読みとばす処理をすることになる

tiny Cの構文解析

- プログラムでは、ASTを作り、それを処理ルーチンに渡す
  - clex.c: 字句解析部分
  - cparser.y : parserのyaccプログラム

字句解析lex.c

- yylexからは、yaccの終端記号が返され、意味値がある場合(終端記号が値を持つ場合)には、yylval.valにセットして返す。

```
int yylex()
{
    int c,n;
    char *p;
again:
    c = getChar();
    if(isspace(c)) goto again;
    switch(c){
    case '+':
    case '-':
    case '*':
    case '>':
    case '<':
    case '(':
    case ')':
    case '{':
    case '}':
    case ']':
    case '[':
    case ';':
    case ',':
    case '=':
    case EOF:
       return c;
```

空白は読み飛ばす

演算子や()などは終端記号として返す

字句解析lex.c

```
 case '"':
       p = yytext;
       while((c = getChar()) != '"'){
           *p++ = c;
       }
       *p = '¥0';
       yylval.val = makeNum((int)strdup(yytext));
       return STRING;
}
if(isdigit(c)){
       n = 0;
       do {
           n = n*10 + c - '0';
           c = getChar();
       } while(isdigit(c));
       ungetChar(c);
       yylval.val = makeNum(n);
       return NUMBER;
}
```

文字列の入力

yytextに入れる

tokenは STRING

意味値を返す この場合は文字列のアドレスを持つNUM

数値が入力された場合

読みすぎた文字をpush back

意味値としてNUMのASTを返す

tokenは、NUMBER

字句解析lex.c

```
if(isalpha(c)){
      p = yytext;
      do {
          *p++ = c;
          c = getChar();
      } while(isalpha(c));
      *p = '¥0';
      ungetChar(c);
      if(strcmp(yytext,"var") == 0)
          return VAR;
      else if(strcmp(yytext,"if") == 0)
          return IF;
      else if(strcmp(yytext,"else") == 0)
          return ELSE;
      else if(strcmp(yytext,"return") == 0)
          return RETURN;
      else if(strcmp(yytext,"while") == 0)
          return WHILE;
      else if(strcmp(yytext,"for") == 0)
```

識別子の場合

名前をyytextに入力

読みすぎた文字をpush back

キーワードかどうか？

キーワードに対応するtokenを返す

プログラミング言語処理

字句解析lex.c

```
        else if(strcmp(yytext,"for") == 0)
            return FOR;
        else if(strcmp(yytext,"println") == 0)
            return PRINTLN;
        else {
            yylval.val = makeSymbol(yytext);
            return SYMBOL;
        }
    }
    fprintf(stderr,"bad char '%c'¥n",c);
    exit(1);
}
```

キーワードでなければ識別子（変数）

意味値としてはSYMのASTを返す

tokenは、SYMBOL

これら以外の文字はエラー！！！！

プログラミング言語処理

構文解析 cparser.y

- parserは、yaccで記述されている

```
%token NUMBER
%token SYMBOL
%token STRING
%token VAR
%token IF
%token ELSE
%token RETURN
%token WHILE
%token FOR
%token PRINTLN
```

まずは、tokenの定義

1文字のtokenは、文字をtokenとして用いる

これらのシンボルは適当な値にdefineされるため、lex.cでつかうことができる

プログラミング言語処理

構文解析 cparser.y

```
%{
#include <stdio.h>
#include "AST.h"
%}

%union {
    AST *val;
}

%right '='
%left '<' '>'
%left '+' '-'
%left '*'

%type<val>  parameter_list block local_vars symbol_list
%type<val>  statements statement expr primary_expr arg_li
%type<val>  SYMBOL NUMBER STRING
```

actionでつかうC言語に必要な定義ファイル

用いる意味値のデータ型はASTvalで指定する

演算子の記号についての結合規則の定義後に書いたほうが優先度が高い

意味値を持つ終端記号、非終端記号のtokenについて、意味値のデータがたを定義する

なお、意味値を持たないtokenについては定義しない

プログラミング言語処理

構文解析 cparser.y

```
%start program

%%
program: /* empty */
       | external_definitions
       ;

external_definitions:
         external_definition
       | external_definitions external_definition
       ;
```

入力全体を指定するtokenを指定する

ここから構文規則が始まる

どのような順番でもいいが、通常、top-downに定義していく

programは空であるか(入力がなし)、もしくは外部定義の列external_definitions

外部定義external_definitionの列の定義

left recursiveになっていることに注意

プログラミング言語処理

構文解析 cparser.y

- 外部定義external_definitionは、関数定義、大域変数の定義、もしくは配列の定義

```
external_definition:
        SYMBOL parameter_list block
      { defineFunction(getSymbol($1),$2,$3); }
      | VAR SYMBOL ';'
      { declareVariable(getSymbol($2),NULL); }
      | VAR SYMBOL '=' expr ';'
        { declareVariable(getSymbol($2),$4); }
      | VAR SYMBOL '[' expr ']' ';'
      { declareArray(getSymbol($2),$4); }
      ;
```

関数定義

大域変数の定義

配列の定義

シンボルを引数にするために、getSymbolを使っていることに注意

プログラミング言語処理

構文解析 cparser.y

- インタープリタ（コンパイラ）とのインタフェース：ここで処理の関数を呼び出す
  - 関数定義の場合は

```
void defineFunction(Symbol *fsym,AST *params,AST *body)
```

fsymに関数名のシンボル、paramsにパラメータのAST、bodyに関数本体のASTを与える。

  - 大域変数宣言の場合は

```
void declareVariable(Symbol *vsym,AST *init_value);
```

vsymに変数名のシンボル、init_valueに初期値のASTを与える。初期値がない場合は、NULL

  - 配列宣言の場合は

```
void declareArray(Symbol *asym,AST *size);
```

asymに配列名のシンボル、sizeにサイズのASTを与える。

構文解析 cparser.y

- パラメータの並びの定義。パラメータは、シンボルの並びを'('と')'ではさんだもの

```
parameter_list:
        '(' ')'
        { $$ = NULL; }
      | '(' symbol_list ')'
        { $$ = $2; }
      ;

symbol_list:
        SYMBOL
        { $$ = makeList1($1); }
      | symbol_list ',' SYMBOL
        { $$ = addLast($1,$3); }
      ;
```

シンボルの並びはシンボルを','で区切ったもの。
まず、最初にmakeList1で最初のリストを作り、それにaddLastで続くシンボルを最後に加えて生成している

構文解析 cparser.y

- blockすなわち複文は、'{'ではじまり、'}'で終る。最初に局所変数の定義 local_varsがあり、文の並びが続くもの。複文を表すASTは、左に局所変数のリスト、右に文のリストをいれたものである。

```
block: '{' local_vars statements '}'
      { $$ = makeAST(BLOCK_STATEMENT,$2,$3); }
      ;
statements:
        statement
       { $$ = makeList1($1); }
      | statements statement
       { $$ = addLast($1,$2); }
      ;
local_vars:
        /* NULL */ { $$ = NULL; }
      | VAR symbol_list ';'
        { $$ = $2; }
      ;
```

BLOCK_STATEMENTのASTを返す

symbolリストと作り方は同じ

変数宣言はなくてもよい

ある場合にはVARから始まる

構文解析 cparser.y

- 文の定義、それぞれの文に対応したASTを作る

```
statement:
        expr ';'
        { $$ = $1; }
      | block
        { $$ = $1; }
      | IF '(' expr ')' statement
       { $$ = makeAST(IF_STATEMENT,$3,makeList2($5,NULL)); }
       | IF '(' expr ')' statement ELSE statement
       { $$ = makeAST(IF_STATEMENT,$3,makeList2($5,$7)); }
      | RETURN expr ';'
       { $$ = makeAST(RETURN_STATEMENT,$2,NULL); }
      | RETURN ';'
       { $$ = makeAST(RETURN_STATEMENT,NULL,NULL); }
      | WHILE '(' expr ')' statement
       { $$ = makeAST(WHILE_STATEMENT,$3,$5); }
      | FOR '(' expr ';' expr ';' expr ')' statement
       { $$ = makeAST(FOR_STATEMENT,makeList3($3,$5,$7),$9); }
      ;
```

式はそれ自身で文になる。ただし、文の終りを表す';'が必要

block文も文

then部とelse部の2つのASTを持つリスト

構文解析 cparser.y

- 式の定義

```
expr:   primary_expr
      | SYMBOL '=' expr
       { $$ = makeAST(EQ_OP,$1,$3); }
      | SYMBOL '[' expr ']' '=' expr
       { $$ = makeAST(SET_ARRAY_OP,makeList2($1,$3),$6); }
      | expr '+' expr
       { $$ = makeAST(PLUS_OP,$1,$3); }
      | expr '-' expr
       { $$ = makeAST(MINUS_OP,$1,$3); }
      | expr '*' expr
       { $$ = makeAST(MUL_OP,$1,$3); }
      | expr '<' expr
       { $$ = makeAST(LT_OP,$1,$3); }
      | expr '>' expr
       { $$ = makeAST(GT_OP,$1,$3); }
      ;
```

変数や配列参照など

変数への代入

配列への代入

2項演算の式については、左には左辺の式のAST、右には右辺の式のASTをいれたASTを作る

2項演算子の優先度については、最初に%right,%leftなどを使って定義してある

構文解析 cparser.y

- primary_expr: 変数や関数呼び出し

```
primary_expr:
        SYMBOL
      | NUMBER
      | STRING
      | SYMBOL '[' expr ']'
        { $$ = makeAST(GET_ARRAY_OP,$1,$3); }
      | SYMBOL '(' arg_list ')'
       { $$ = makeAST(CALL_OP,$1,$3); }
      | PRINTLN  '(' arg_list ')'
       { $$ = makeAST(PRINTLN_OP,$3,NULL); }
      ;
arg_list:
        expr
       { $$ = makeList1($1); }
      | arg_list ',' expr
       { $$ = addLast($1,$3); }
      ;
```

関数呼び出しでは、左に関数名、右に引数のならびのリストを持つCALL_OPのASTノードを作る

システム関数println の呼び出し

構文解析 cparser.y

- 終わりに、clex.cをincludeしておく。

```
%%

  #include "clex.c"
```

これで構文規則の記述を終了

このようにすることのメリットは、clex.cのなかで、 tokenの名前をマクロで定義されているものとして、使うことができる点

yyerror

- yaccで生成される構文解析ルーチンでは、エラーを起こした時に、つまり間違った構文が入力されたときに、yyerrorという関数が呼び出されるようになっている。
  - このプログラムでは、yyerrorは以下のように、メッセージをプリントしてプログラムを停止する、簡単なものにしている。

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

プログラムではエラーの処理については考慮されていない。
本格的なプログラムにするには、エラー処理について考慮する必要がある

コンパイルの仕方

- cparser.yから、yaccを使ってparserを生成する。

```
% yacc cparser.y
```

- 生成されたプログラムは、y.tab.cになっているので、これを適当な名前 (cparser.c)に変えて、Cコンパイラで、コンパイルする。

```
% mv y.tab.c cparser.c
% cc -c cparser.c
```

- なお、clex.cはcparser.cにincludeされているので、別にコンパイルする必要はない。

インタプリター

- tiny-Cのインタープリタを作ってみることにする。
- まず、式の実行から考えてみよう。変数を考えなければ、大体は式の評価でつくったインタープリタと同じである。その後に、言語の重要な機能である関数について考えてみることにする。
- 説明するプログラムは、以下にある。
  - interp.h : インタプリタのheader
  - interp_expr.c : インタプリタの式の処理
  - interp.c : インタプリタの関数、文の処理

変数の扱い

- 変数の値を格納しておくためには、シンボル構造体のvalのフィールドにいれておく。
  - シンボル構造体は以下のようになっていた。

```
typedef struct symbol {
    char *name;
    int val;   /* ← これを用いる */
    AST *func_params;
    AST *func_body;
} Symbol;
```


式の評価

```
int executeExpr(AST *p)
{
    if(p == NULL) return 0;
    switch(p->op){
    case NUM:
       return p->val;
    case SYM:
       return getValue(getSymbol(p));
    case EQ_OP:
       return setValue(getSymbol(p->left),executeExpr(p->right));
    case PLUS_OP:
       return executeExpr(p->left) + executeExpr(p->right);
    case MINUS_OP:
       return executeExpr(p->left) - executeExpr(p->right);
    case MUL_OP:
       return executeExpr(p->left) * executeExpr(p->right);
    case LT_OP:
       return executeExpr(p->left) < executeExpr(p->right);
    case GT_OP:
```

これはどうなっていればいいのか？

変数の値の参照

- 関数を考えなければこれでいい。

```
int getValue(Symbol *var)
{
    return var->val;
}

int setValue(Symbol *var,int val)
{
    var->val = val;
    return val;
}
```

単なる式を評価するだけならば、以上のコードで十分であるが、実際はもうすこし仕掛けが必要となる。それは関数のパラメータ引数や局所変数があるからである

次回は

- tiny Cのインタープリタ
- 関数と環境

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。