情報科学実験II 第2回

コンパイルの4つのステップ

1、字句解析(lexical analysis)
2、構文解析(syntax analysis) ⇐ 今日はここに関係する話
3、意味解析(semantic analysis)
4、コード生成(code generation)

目標
・シンタックス（文法）ベースの考え方に慣れる
・yaccの基本的な使い方を知る

文法と構文解析

日本語文「日本の四季は美しい」

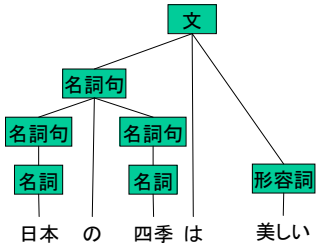
⇩ この文の意味を解釈するには・・・・

1、「日本」、「四季」、「美しい」の意味を辞書から探す
2、「日本の四季」を1つまとまりとする
3、A＋「は」＋Bという語順が、AはBという性質を持つ
という意味であるということから文を解釈する

文法とは

文の分解の仕方（読むとき）＝文の構成の仕方（書くとき）

例）



文法とは

"文とは名詞句＋「は」＋形容詞である"と読む

文法の例）

文→名詞句「は」形容詞
2つ { 名詞句→名詞句「の」名詞句
名詞句→名詞
名詞→「日本」
名詞→「四季」
形容詞→「美しい」

再帰的定義：名詞句の定義に名詞句が含まれる「僕のお父さんの部屋の時計の・・・・」

「は」「日本」などの文に現れる記号と、「文」「名詞」などの文に現れない記号があることに注意。前者を終端記号、後者を非終端記号という。

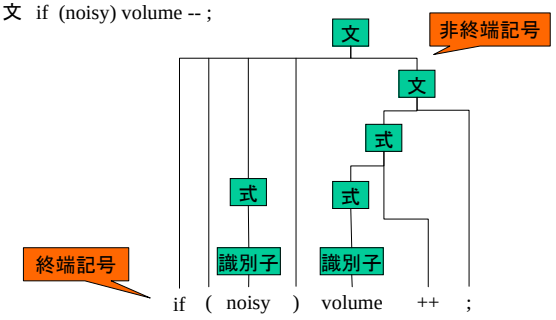
※注意
文法的に正しくても意味の通る文になるとは限らない。
「四季の日本は美しい」という文も文法的には正しい。

⇩

文法は語の関連付けの仕方を表すだけ

C言語風ifの文法と構文木

文　if (noisy) volume -- ;



C言語の文法

文法の例）　意味的にひとまとまりになる字句の列をひとまとめにして表す

字句解析によって取り込める

非終端記号	文	式	識別子					
終端記号	if	(	)	;	++	noisy	volume	
生成規則	文→if (式) 文							

左辺　右辺

文→式;
式→式++
式→識別子
識別子→noisy
識別子→volume

開始記号　文

C言語のif文の意味

式　文　文

構文：　`if (`*expr*`)` *stat*

意味：　まず*expr*を評価し、その値が真ならば*stat*を実行する

exprの値=0なら偽、それ以外なら真

このような意味がわかって、はじめて機械語プログラムに翻訳できる

⇩

コンパイルのためには文法規則から
文の構成や構文木を求めることが必要

構文解析器の自動生成ツール

yaccにおける文法の書き方

左辺　　右辺

文→if (式) 文
文→式;
式→式++
式→識別子
識別子→noisy
識別子→volume

⇒

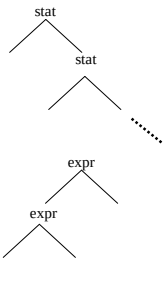
```
stat:  IF LPAR expr RPAR stat
     | expr SC
     ;

expr:  expr INC
     | id
     ;

id:    NOIZY
     | VOLUME
     ;
```

・終端記号は大文字で、非終端記号は小文字で書くのが慣わし
・左辺が同じ規則は右辺を'|'でまとめてよい
・規則の最後に;をつける

再帰性

```
stat:  IF LPAR exp RPAR stat
     ;
```

右再帰・・・左辺と同じ非終端記号が右辺の最後に来る




```
expr:  expr INC
     ;
```

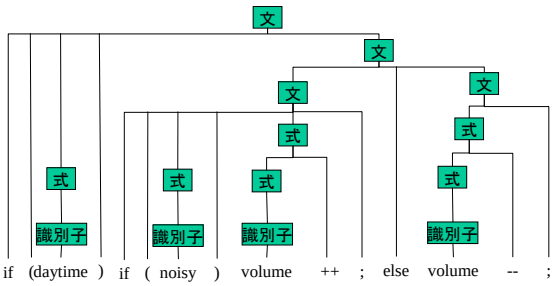
左再帰・・・左辺と同じ非終端記号が右辺の最初に来る

再帰性により、ある非終端記号が任意の回数繰り返されることを許すような規則が表現できる

elseのやっかいな問題 -あいまい性-

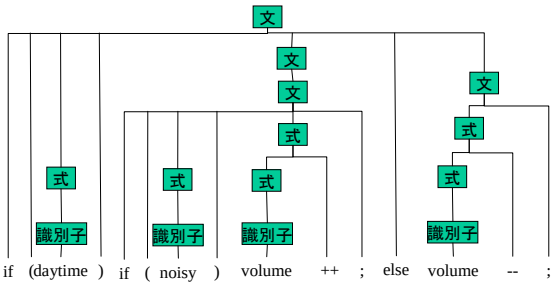
構文木は1通りに定まるとは限らない

例） if (daytime) if (noisy) volume++; else volume--;



※elseが内側のifと組になる場合

例） if (daytime) if (noisy) volume++; else volume--;



※elseが外側のifと組になる場合

elseの解釈の違い

練習1　条件により実行されるstatがもしあれば以下の表中に埋めよ。
なければ無しと記入せよ。

内側のifと組になる場合

	daytime=真	daytime=偽
noisy=真	volume++;	
noisy=偽		

外側のifと組になる場合

	daytime=真	daytime=偽
noisy=真		
noisy=偽		

練習2（あいまい性）
先のyaccの文法を、elseが使えるように拡張してみよ。C言語と同様内側のifとelseが組になるようにせよ。

さらにもう1つのコンパイラのコンパイラ

yacc
–Yet Another Compiler compiler –

・・・構文解析プログラムを自動生成するツール

ファイル名は xxxx.y

定義セクション
ルールセクション
Cコードセクション

```
%{
    この部分は構文解析プログラムの先頭に挿入される。
    必要なヘッダファイルなどを書く。
%}
    ここで終端記号を宣言

%%
    ここから文法規則を書く

%%
    この部分は構文解析プログラムの最後に挿入される。
    例えば、字句要素（解析器から見ると終端記号＝トークン）
    を取り込む関数(yylex)を書く
```

yaccに生成された構文解析プログラムの例

yaccが勝手に割り当てる

```
#include     <stdio.h>
#define      NUMBER 257 終端記号はこのように整数定数として定義される
int yylval;   yylexから返る終端記号の「値」yylvalを型と一緒に宣言

int yylex()  字句解析関数・・・・yaccプログラムの中に自分で書く。lexに任せるなら不要。
             ・・・・
}

void yyerror(msg)  構文解析に失敗したときの処理。エラーメッセージの出力など。
             char *msg;
{
             …
}

int main()   メイン
{
             …
}

yyparse()    パーザ本体
{            この中からyylexとyyerrorが呼ばれる
             …
}
```

簡単なyaccプログラムの例　-if-

```
%{
#include <stdio.h>
%}                                  ヘッダファイル

%%
input: /* empty*/                   コメント
             | input line           「inputとはlineの繰り返し」
             ;                      2行以上の入力に対応

line:        '¥n'                   「lineとは改行もしくはstatに
             | stat '¥n'    { printf("%d¥n", $1); }   改行が続くもの」
             ;

stat:   'i' '(' expr ')' stat { printf("if¥n"); }
             | 'i' '(' expr ')' stat 'e' stat { printf("ifelse¥n"); }
             | expr ';'
             ;

expr:   id
        | id '+'
        ;

id:     'n'
        | 'v'

%%
```

exp=expression 式の意味

非終端記号ごとの文法規則。右辺に非終端記号があると、解析器はさらにそれを左辺にもつ規則の適用を試みることに注意。

{}内はルールにマッチしたときのアクション(Cのプログラムで書く)。コンパイラはここで解析木を出力するが、ここでは代わりにマッチしたstatement名を出力している。

マッチ＝左辺が全て満たされること

キーワードifの略。ここでは字句解析を簡単化して1文字のみ取り込むようにしているため、キーワードも1文字。

構文解析関数はyyparse

```
int main()
{
        return yyparse();
}

int yyerror(const char *s)
{
        printf("ERROR: %s¥n", s);
        return 0;
}


int yylex()
{
        int c = getchar();
        return c;
}
```

main、エラー処理yyerror、字句要素を取り込む関数であるyylexを定義。yylexは、呼ばれるたびに、字句要素を返すように書く。ここでは、単に1文字ずつ読み込んでいる。

yaccでコンパイル　$ yacc xxxx.y
パーザのCプログラム xxx.tab.cを自動生成
xxx.cをコンパイルするとパーザ完成　$gcc y.tab.c

```
#include     <stdio.h>  ここに%{から%}までの部分が挿入されている

#define      NUMBER 257     宣言した終端記号があれば、整数定数として定義される
int yylval;

int yylex()
             ・・・・
}

void yyerror(msg)
             char *msg;
{
             …
}

int main()
{
             …
}

yyparse()    パーザ本体
{            この中からyylexとyyerrorが呼ばれる
             …
}
```

練習3　if　else 構文の例を実際にyaccで実行してみよ。様々な入力について、正しく解析されるかどうか確認せよ。解析に失敗するstatは、なぜ失敗するかそれぞれ理由をのべよ。

式を入力してCtrl-d

入力の例）
n;
n
(n);
i(n)n;
i(n)i(v);
i(n)i(n)v;
i(n)n;n;
i(n)v;en;
i(n)v;ei(n)v;ei(v)n;
i(n)v;ei(n)v;ei(v)n;n;
i(n) i(v) i(n)v;en;ev;en;

練習4　課題3で失敗したstatについて、複数のstatが使えるように文法を変更・拡張せよ。

yaccプログラム（加算電卓）の例

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
%}

%token      NUM

%%
input: /* empty*/
        | input line
        ;

line:   '¥n'
        | exp '¥n'      { printf("%d¥n", $1); }
        ;

exp: NUM
        | exp '+' exp  { $$ = $1 + $3; }
        ;

%%
```

ヘッダファイル

終端記号などの宣言

文法の終端記号（トークン）を定義しておくと、以下の規則で使える。

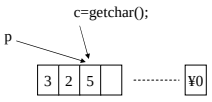
exp=expression 式の意味

$xはyylexが戻した終端記号の値yylvalを受け取る。何を戻すかはyylexの書き方による。このプログラムでは十進数か演算子がint(char)で戻っている。$2には'+'が入っている点に注意。$$は左辺のexpの値を次の規則に返すための変数で、何を返してもよいが、何も書かなければ$1が入る。型はintである。
　終端記号の種類は、対応する整数定数がyylexの「関数の返値として」返される。yylvalとは違う点に注意。

```
int main()      /* main:yacc */
{
        return yyparse();
}

int yyerror(const char *s)      /* error */
{
        printf("ERROR: %s¥n", s);
        return 0;
}

int yylex()     /* 字句解析ルーチンはyylexという関数にする */
{
        int c = getchar();
        if (isdigit(c)) {
                char buff[128];
                char *p = buff;
                do {
                        *p++ = c;
                        c = getchar();
                } while (isdigit(c));
                ungetc(c, stdin);
                *p = '¥0';
                yylval = atoi(buff);
                return NUM;
        }
        return c;
}
```

構文解析関数はyyparse

ここでは、[0-9]の列が来たときに、それを10進整数に変換して構文解析器yyparseに返している。



この場合、トークンの種類はNUMだけ。yylexはトークンをみつけたら、その「値」をグローバル変数yylvalに入れNUMを「関数の値として返す」（つまり、2つ返す）。yylvalは、順に$xに入れられる。

練習5（yaccの練習）
　加算電卓のyaccプログラムからパーザを生成し、様々な入力に対してパーザの動作を確認せよ。

例）
3
2+3
4+2+5
4-2
3*2
・・・・

練習6（yaccの練習）
　加算電卓に減算を追加してみる。以下のように規則を変更して動作を確認してみよ。どのような場合に誤った結果となるか。

```
exp: NUM
        | exp '+' exp           { $$ = $1 + $3; }
        | exp '-' exp           { $$ = $1 - $3; }
        ;
```

例）
2-3
1-2
1-1-1
2-1-1
4+2-5
4+2+3
・・・・

演算子の結合性

被演算子　OP　被演算子　OP　被演算子

?　　?

加算の場合
2+3+4・・・・2+3と3+4いずれでも結果はかわらない。＋は左右どちらからでも評価できる。

減算の場合
2-3-4・・・・2-3を求めてそこからさらに4をひく。ーは右から評価してはいけない。

yaccの解析アルゴリズムでは、基本は右結合性（同じ演算子が並んだら右側の被演算子を先に処理する）になる

なぜか？

yaccの構文解析法　LALR(1)

- Look Ahead Left-to-right scanning
right most derivation in Reverse (1)-

基本動作：入力を左端から読んでいって（シフト）、規則の右辺の条件がそろったら左辺の非終端記号に置き換える（還元）

例）　4-3+2

⇧　NUMを発見 ⇨ expに還元してスタックに積む

⇧　'-'を発見 ⇨ スタックに積む

⇧　NUMを発見 ⇨ expに還元してスタックに積む

スタックの中身

exp
-
exp

exp '-' exp　という規則にマッチしたので左辺のexpに還元・・・とはしない

yaccの構文解析法　LALR(1)

LALR(1)の場合、還元せずにできるだけシフトする（その方がうまくいく文法が多い）。

例）　4-3+2

⇧　NUMを発見 ⇨ expに還元してスタックに積む

スタックの中身

exp
-
exp

還元せずにそのままシフトして+を読む

yaccの構文解析法　LALR(1)

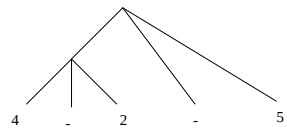
例）　4-3+2

⇧　+を発見 ⇨ スタックに積む

スタックの中身

+
exp
-
exp

左結合性にするには・・・・



となるようにすればよい。

練習7（左結合性）
文法を変更して、減算が正しく動作するようにしてみよ。

例）
2-3
1-2
1-1-1
2-1-1
4+2-5
4+2+3
4+2-5+3-1
・・・・

練習6の解説

```
%%
input: /* empty*/
        | input line
        ;

line:   '¥n'
        | exp '¥n'  { printf("%d¥n", $1); }
        ;

exp: NUM
        | exp '+' NUM       { $$ = $1 + $3; }
        | exp '-' NUM       { $$ = $1 - $3; }
        ;

%%
```

演算子の優先順位

3-4-2 ⇒ 同じ演算子が並ぶ場合は結合性を決めて解決
異なる演算子が並んだら？

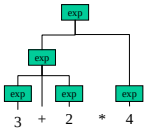
さらに四則演算に拡張してみる

```
exp: NUM
        | exp '+' NUM        { $$ = $1 + $3; }
        | exp '-' NUM        { $$ = $1 - $3; }
        | exp '*' NUM        { $$ = $1 * $3; }
        | exp '/' NUM        { $$ = $1 / $3; }
```

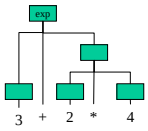
3*4+2 ⇒ これだとタマタマうまくいく（左結合性を入れた副産物）

3+4*2 ⇒ これだとうまくいかない

例） 3+2*4の場合



こうなってしまう。加減算と乗除算の優先度が考慮されていない。



こんな感じになってほしい

課題1 (四則演算電卓)

四則演算の規則を足した前ページの例を確認してみよ。さらに練習5の電卓を四則演算が使えるように修正せよ。様々な入力に対して正しい結果が得られることを確認せよ。

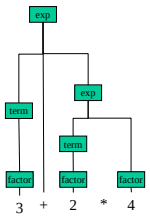
例）
3+2
4-5
2*3
8/3
3+2*4
2*4/5
2+2*4/5
3+2*4/5-2
・・・・

課題1の解説

数を因子(factor)、乗除算演算子を間に持つような式を項(term)、と名付けて、規則を書き直す。

```
exp: NUM
        | term                { $$ = $1; }
        | exp '+' term        { $$ = $1 + $3; }
        | exp '-' term        { $$ = $1 - $3; }
        ;
term: factor
        | term '*' factor     { $$ = $1 * $3; }
        | term '/' factor     { $$ = $1 / $3; }
        ;
factor: NUM
        ;
```

例） 3+2*4の場合



課題2 (括弧つき四則演算)

課題1の文法を、括弧つきの四則演算に拡張せよ。

例） 2*(3+1*2)+1　　3*(1+2)*1+2*1

課題3 (階乗・冪乗)

課題3の文法を、階乗(!)と冪乗(^)が使えるように拡張せよ。演算子の優先順位は、階乗 ＞ 冪乗 ＞ 乗除 ＞ 加減である。

例） 3+2*3^2-1　2-2^3!-2などが優先度の通り正しく解析できるかどうか

ヒント アクションで階乗と冪乗を記述する。必要に応じて関数を定義する。

発展課題 (Google電卓)

Google電卓のサブセットを可能な限り作成してみよ。

場合により、$xで扱う型を変更する必要あり（デフォルトはint）。そのためには、YYSTYPEマクロを変更する。Cのコードに、

```
#undef YYSTYPE
#define YYSTYPE double
```

と入れればよい。その他、以下を参照のこと。

参考　bisonの日本語マニュアル

http://www.mi.s.osakafu-u.ac.jp/~kada/course-kitami/j3_03/bison_j.html