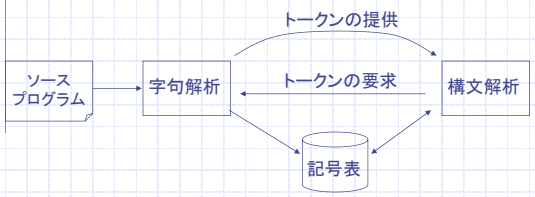
コンパイラ理論 4
構文解析 導入

櫻井彰人

字句解析と構文解析



なぜ分けるか?

- 字句解析を構文解析から分ける理由:
  - 設計が単純になる
  - 効率(速度等)の向上が図れる
  - 可搬性がます
- 字句解析・構文解析それぞれによいツールが存在する

トークン・字句・パターン
(Tokens, Lexemes, Patterns)

- トークンは、キーワード(if, for, long,...)、演算子(+,*,...)、識別子、定数、文字列、区切り記号を含む、字句が属すクラスのことをいう
- 字句は、文字のある列であって、ソースプログラム内で意味をもつ最小の単位
- パターンは、(lexで用いるが)あるトークンの生成規則

属性 Attributes

- 属性は、トークンのもつ情報。変数、定数、配列、キーワード、演算子、、、
- 字句解析は、通常、属性は一個しか与えない(構文解析で、いくつも追加される)

文字列と言語

- アルファベット Alphabet – 記号の有限集合 (例: ASCII, JIS漢字コード, トークの集合)
- 文字列 String – アルファベット内の記号の有限列
- 言語 Language – あるアルファベットから作られる文字列の集合
- 文字列に関する用語: 接頭辞 prefix; 接尾辞 suffix; 部分文字列 substring;

構文解析器 パーサー Parser

- 字句解析器からトークン列を受け取り(通常は、一時に1トークン)
- そのトークン列が、所定の文法で生成可能かどうかを調べる
- もし構文上の誤りがあればそれを報告する (可能な限り修復する)

誤り

- 字句の誤り lexical errors (例: 綴り誤り)
- 構文の誤り syntax errors (例: 括弧が対応しない、セミコロン忘れ)
- 意味の誤り semantic errors (例: 型誤り)
- 論理的な誤り logical errors (例: 無限ループ)

エラーの取り扱い

- エラーは、明確かつ正確に報告する
  - 実はこれが難しい
  - 現象と原因との「距離」が離れている
  - 特に、抽象度のレベルが違う
- できるだけ回復する
  - そこで止まらない、先へ進む
- しかし、エラー回復が下手だと、エラーの山が築かれる

エラー回復

定義がないと分かりませんね

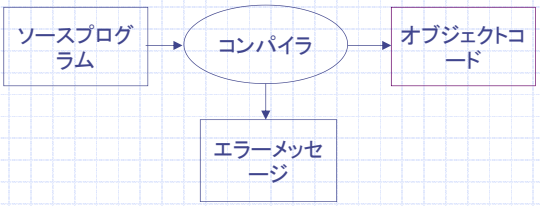
- パニックモード panic mode: 最近のトークンに対応するトークンがでてくるまでトークンを読み捨てる
  - 実は、これができるのは、文脈自由文法の性質による
- 句 phrase レベルの回復: 非終端記号を読替えて、構文解析が継続できるようにする
- エラーの生成: 文法に、予想されるエラーを生成するような文法規則を追加する
- 全体的な変換: 複雑なアルゴリズムで、コスト最小の変更で、構文解析可能なコードに変換する

コンパイラ・コンパイラの目的

- コンパイラ・コンパイラ: 言語仕様からその言語のコンパイラを作るコンパイラ、ということは、
  - 「コンパイラ記述用の言語を用いて書いたプログラム」(コンパイラに決まっている)をコンパイルするプログラム。
- なぜこんなややこしいことを考えたのか?
  - コンパイラを作るのは大変な作業
    - FORTRAN I のコンパイラ開発に何年かかったと思う?
  - コンパイラを書くための言語があったらいいなあ

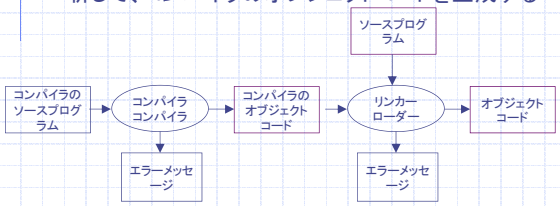
コンパイラ

- ソースプログラムを解析して、オブジェクトコードを生成する



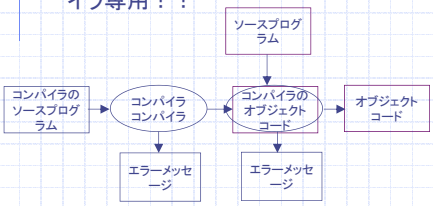
コンパイラ･コンパイラ

- コンパイラ: ソースプログラムを解析して、オブジェクトコードを生成する
- コンパイラ･コンパイラ: コンパイラのソースコードを解析して、コンパイラのオブジェクトコードを生成する



コンパイラ･コンパイラ

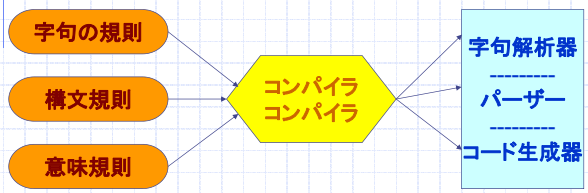
- コンパイラ･コンパイラ: コンパイラのソースコードを解析して、コンパイラのオブジェクトコードを生成する
- 当然、コンパイラのソースコードを各言語は、普通のプログラムのソースコードを書く言語とは異なる。コンパイラ専用！！



コンパイルのフェーズ

- コンパイルのフェーズ(おおまか):
  - 字句解析 lexical analysis
  - 構文解析 syntax analysis
  - 意味解析 semantic analysis
  - 最適化 optimization
  - コード生成 code generation

コンパイラコンパイラ



Yacc プログラムの例

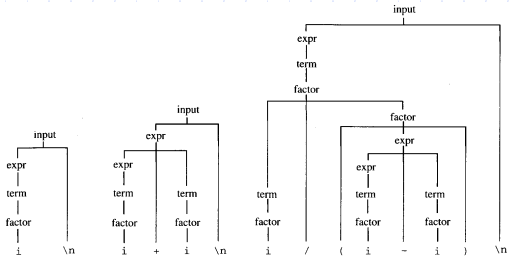
プログラム 2.1

```
1.  /* プログラム2.1(21ページ) */
2.  /*  Yaccで記述した式の定義  */

3.  %%

4.  input  : expr '¥n' ;
5.  expr   : expr '+' term | expr '-' term | term ;
6.  term   : term '*' factor | term '/' factor | factor ;
7.  factor : 'i' | '(' expr ')' ;

8.  %%

9.  yylex()
10. {
11.   return getchar();
12. }
```

```
1.  /* プログラム2.1(21ページ) に num digit を追加 */
2.  /*  Yaccで記述した式の定義  */     コメント

3.  %%   構文規則の記述が始まることを示す

    非終端記号 input を定義する構文規則. 入力データ (input) は, 非終端記号 expr によって規定される文字列のあとに, 改行(¥n)が続いたもの
    識別子は, 特別な値をもたせる場合を除いて, 宣言する必要はない
    開始記号の宣言をしない限り, 最初の構文規則によって定義する非終端記号が開始記号として扱われる
4.  input  : expr '¥n' ;   選択肢の間は, "|" で区切る. '+' や '-' は終端記号
                           実は, yylex が返す値.
                           Yaccプログラムにとっての入力データはトークンの列である
5.  expr   : expr '+' term | expr '-' term | term ;
6.  term   : term '*' factor | term '/' factor | factor ;
7.  factor : num | '(' expr ')' ;
8.  num    : digit | num digit ;
9.  digit  : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;

10. %%   構文規則の定義が終わり, 関数定義が始まることを表す

11. yylex()
12. {
13.   return getchar();
14. }
```

getchar() を用いて, 標準入力から1文字を読み込み,その文字コードをトークンとして返す関数 yylex() の定義

構文解析木の例



再帰下降法(recursive descent)

- Pascal のコンパイラで採用
  - 再帰下降法が使えるような言語仕様となっている
- 例

```
expr → term { ( + | – )  term }
term → factor { ( * | / )  factor }
factor → id | const | ( expr )
```

```
void Expr(void)
{
  Term();
  while (NextSym =='+' || NextSym =='-' ){
    int Op = NextSym;
    NextSym = yylex() ;
    Term() ;
    printf(" %c " , Op); }
}
```

次のトークンが一個先読みされている

従って、次のトークンを一個先読みする必要がある

再帰下降法

```
expr → term { ( + | – )  term }
term → factor { ( * | / )  factor }
factor → id | const | ( expr )
```

```
void Term(void)
{
  Factor();
  while (NextSym =='*' || NextSym =='/' ){
    int Op = NextSym;
    NextSym = yylex() ;
    Factor() ;
    printf(" %c " , Op); }
}
```

```
void Factor(void)
{
  switch ( NextSym.attribute )  /* いんちき */
   {
     case id: Id(); break;
     case const: Const(); break;
     case '(' :
       NextSym = yylex() ;
       Expr() ;
       if ( NextSym == ')' ) return 0;
       else error ("should be right paren");
    }
}
```

Pascal compiler の一部

```
procedure expression;
  var lattr: attr; lop: operator; typind: char; lsize: addrrange;

  procedure simpleexpression(fsys: setofsys);
    var lattr: attr; lop: operator; signed: boolean;

    procedure term(fsys: setofsys);
      var lattr: attr; lop: operator;

      procedure factor(fsys: setofsys);
        var lcp: ctp; lvp: csp; varpart: boolean;
            cstpart: set of 0..58; lsp: stp;
      begin
        if not (sy in facbegsys) then
          begin error(58); skip(fsys + facbegsys);
            gattr.typtr := nil
          end;
        while sy in facbegsys do
          begin
            case sy of
(*id*)        ident:
                begin searchid([konst,vars,field,func],lcp);
                  insymbol;
                  if lcp^.klass = func then
                    begin call(fsys,lcp); gattr.kind := expr end
                  else
                    if lcp^.klass = konst then
                      with gattr, lcp^ do
                        begin typtr := idtype; kind := cst;
                          cval := values
                        end
                    else
                      begin selector(fsys,lcp);
                        if gattr.typtr<>nil then(*elim.subr.types to*)
                          with gattr,typtr^ do(*simplify later tests*)
                            if form = subrange then
                              typtr := rangetype
                      end
          end;
```

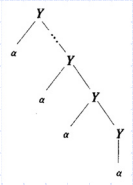
```
(*cst*)   intconst:
          begin
            with gattr do
              begin typtr := intptr; kind := cst;
                cval := val
              end;
            insymbol
          end;
        realconst:
          begin
            with gattr do
              begin typtr := realptr; kind := cst;
                cval := val
              end;
            insymbol
          end;
        stringconst:
          begin
            with gattr do
              begin
                if lgth = 1 then typtr := charptr
                else
                  begin new(lsp,arrays);
                    with lsp^ do
                      begin aeltype := charptr; form:=arrays;
                        inxtype := nil; size := lgth*charsize
                      end;
                    typtr := lsp
                  end;
                kind := cst; cval := val
              end;
            insymbol
          end;
(*(*)     lparent:
          begin insymbol; expression(fsys + [rparent]);
            if sy = rparent then insymbol else error(4)
          end;
(*not*)   notsy:
          begin insymbol; factor(fsys);
            load; gen0(19(*not*));
            if gattr.typtr <> nil then
              if gattr.typtr <> boolptr then
                begin error(135); gattr.typtr := nil end;
          end;
```

```
(*[*)     lbrack:
          begin insymbol; cstpart := [ ]; varpart := false;
            new(lsp,power);
            with lsp^ do
              begin elset:=nil;size:=setsize;form:=power end;
            if sy = rbrack then
              begin
                with gattr do
                  begin typtr := lsp; kind := cst end;
                insymbol
              end
            else
              begin
                repeat expression(fsys + [comma,rbrack]);
                  if gattr.typtr <> nil then
                    if gattr.typtr^.form <> scalar then
                      begin error(136); gattr.typtr := nil end
                    else
                      if comptypes(lsp^.elset,gattr.typtr) then
                        begin
                          if gattr.kind = cst then
                            cstpart := cstpart+[gattr.cval.ival]
                          else
                            begin load; gen0(23(*sgs*));
                              if varpart then gen0(28(*uni*))
                                else varpart := true
                            end;
                          lsp^.elset := gattr.typtr;
                          gattr.typtr := lsp
                        end
                      else error(137);
                  test := sy <> comma;
                  if not test then insymbol
                until test;
                if sy = rbrack then insymbol else error(12)
              end;
```

```
            if varpart then
              begin
                if cstpart <> [ ] then
                  begin new(lvp,pset); lvp^.pval := cstpart;
                    lvp^.cclass := pset;
                    if cstptrix = cstoccmax then error(254)
                    else
                      begin cstptrix := cstptrix + 1;
                        cstptr[cstptrix] := lvp;
                        gen2(51(*ldc*),5,cstptrix);
                        gen0(28(*uni*)); gattr.kind := expr
                      end
                  end
              end
            else
              begin new(lvp,pset); lvp^.pval := cstpart;
                lvp^.cclass := pset;
                gattr.cval.valp := lvp
              end
          end
      end (*case*) ;
      if not (sy in fsys) then
        begin error(6); skip(fsys + facbegsys) end
    end (*while*)
  end (*factor*) ;

begin (*term*)
```

左再帰と右再帰

左再帰: X → α | Xα

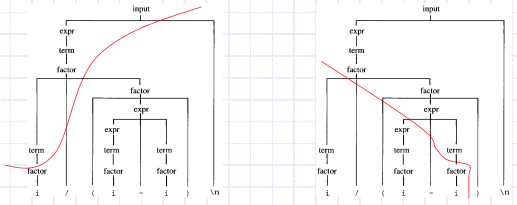


右再帰: Y → α | αY

- 左再帰: 再帰下降法では無限ループになる
- 右再帰: 上昇法では、スタックが深くなる

LRとLL

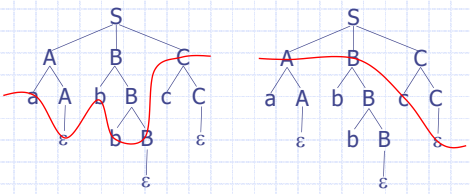
- Yacc は LR構文解析
  - Left to right Rightmost derivation
- 再帰下降は LL構文解析
  - Left to right Leftmost derivation



最左導出と最右導出

- leftmost and rightmost derivation
- 文法 G = ({S, A, B, C}, {a, b, c}, S, P)
  ただし P = {S→ABC, A→aA, A→ε, B→bB, B→ε, C→cC, C→ε}.
- 展開（導出）中に、展開すべき非終端記号が複数個ある。その選び方で、導出過程が異なる
  S⇒ABC⇒aABC⇒aABcC⇒aBcC⇒abBcC⇒abBc⇒abbBc⇒abbc
- 最も左の変数を展開すると
  S⇒ABC⇒aABC⇒aBC⇒abBC⇒abbBC⇒abbC⇒abbcC⇒abbc
- 最も右の変数を展開すると
  S⇒ABC⇒ABcC⇒ABc⇒AbBc⇒AbbBc⇒Abbc⇒aAbbc⇒abbc

- 一般には、展開する変数の選び方で、結果は大きく異なる
- しかし、（曖昧でない）文脈自由文法では、導出結果は同じ。



最左導出
(leftmost derivation)

最右導出
(rightmost derivation)