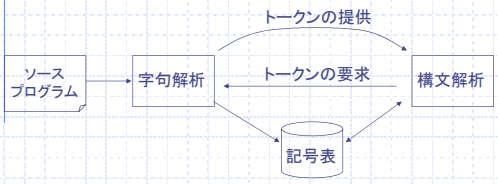
# コンパイラ理論 4 構文解析1

2012.05.18

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

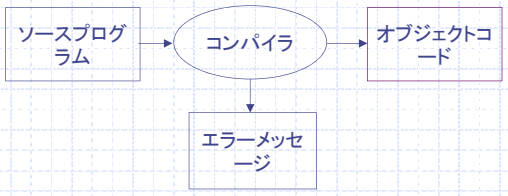
- パニックモード panic mode: 最近のトークンに対応するトークンがでてくるまでトークンを読み捨てる
  - 実は、これができるのは、文脈自由文法の性質による
- 句 phrase レベルの回復: 非終端記号を読替えて、構文解析が継続できるようにする
- エラーの生成: 文法に、予想されるエラーを生成するような文法規則を追加する
- 全体的な変換: 複雑なアルゴリズムで、コスト最小の変更で、構文解析可能なコードに変換する

## 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

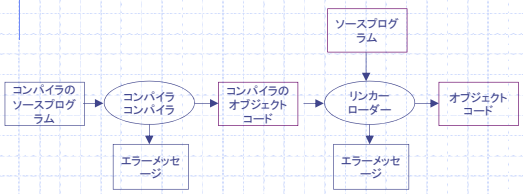
## コンパイラ

- 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



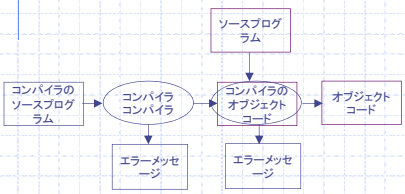
## コンパイラ・コンパイラ

◆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

◆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

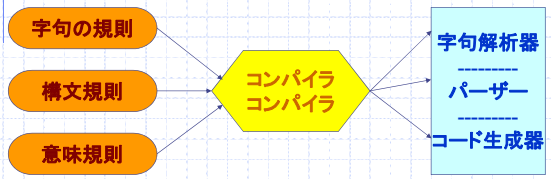



## コンパイルのフェーズ

◆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



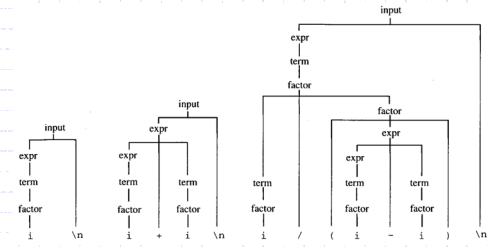

## Yacc プログラムの例

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. {                      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
13.   return getchar();
14. }
```

## 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

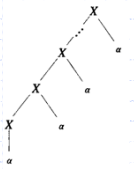
## 再帰下降法

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

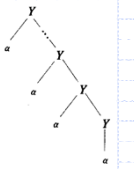
```
void Term(void)
{
  Factor();
  while (NextSym =='*' || NextSym =='/' ){
    int Op = NextSym;
    NextSym = yylex() ;
    Factor() ;
    printf(" %c " , Op); }
}
```

```
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");
   }
}
```

## 左再帰と右再帰




左再帰: $X \rightarrow \alpha \mid X\alpha$

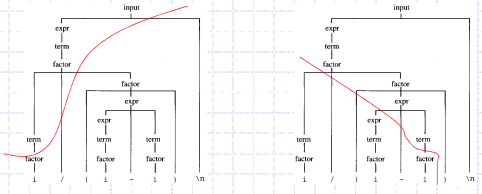



右再帰: $Y \rightarrow \alpha \mid \alpha Y$

- 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




## 復習

0型文法
　句構造文法(phrase structure grammar)
1型文法
　文脈依存文法(context sensitive G)
2型文法
　文脈自由文法(context free grammar)
3型文法
　正規文法(regular grammar)

（3）文法の記述
バッカス記法(Backus Naur Form：BNF）

記号1：：＝記号2　記号3
（記号1は記号2と記号3の
列に書き換えられる）
記号1：：＝記号列1｜記号列2
（記号1は記号列1か記号列2
に書き換えられる）
拡張バッカス記法
{　}　…0回以上の繰り返し

## 正規表現

- 3型文法を記述する方法の一つ
  - BNFでも、勿論、できる
  - 例: a ( b | c ) * (d | e)+ f
- JavaCC で使うメタ記号
  - |　選択
  - ( ) *　0回以上の繰り返し
  - ( ) +　1回以上の繰り返し
  - ( ) ?　あってもなくても可
  - [ ]　どれか一つ (集合)
  - ^　(右に書いてあるもの)以外
  - < >　他の規則で定義されたトークン

## ちょっと詳細に

| | | |
|---|---|---|
| "..." | 引用符内に書かれた文字（列）と一致 | Javaの文字列と同じ記法 |
| \| | または | 演算子としての優先順位はもっとも低い |
| [...] | 文字クラス | 文字クラス内でのみ使えるメタ文字がある |
| ~[...] | 補集合文字クラス | 内部は文字クラスと同じ構文 |
| - | 範囲 | 文字クラス内でのみ（例："a"-"z"） |
| , | 列挙 | 文字クラス内でのみ（例："a", "b"） |
| (...)? | 省略可能 | 丸括弧が常に必要 |
| (...)* | 任意回繰り返し | 丸括弧が常に必要 |
| (...)+ | 1回以上の繰り返し | 丸括弧が常に必要 |
| (..){n} | n回の繰り返し | 丸括弧が常に必要 |
| (..){n,m} | n回からm回の繰り返し | 丸括弧が常に必要 |