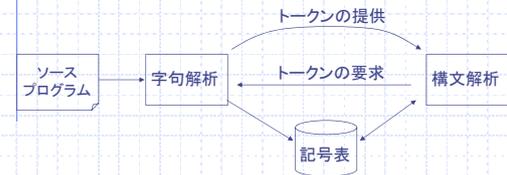


コンパイラ理論 4 構文解析 導入

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
 - 可搬性がまず
- ◆ 字句解析・構文解析それぞれによりツールが存在する

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

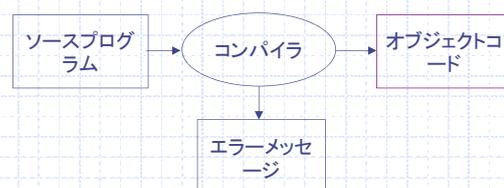
- ◆ パニックモード panic mode: 最近のトークンに対応するトークンがでくるまでトークンを読み捨てる
 - 実は、これができるのは、文脈自由文法の性質による
- ◆ 句 phrase レベルの回復: 非終端記号を読替えて、構文解析が継続できるようにする
- ◆ エラーの生成: 文法に、予想されるエラーを生成するような文法規則を追加する
- ◆ 全体的な変換: 複雑なアルゴリズムで、コスト最小の変更で、構文解析可能なコードに変換する

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

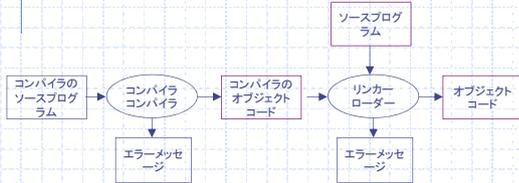
コンパイラ

- ◆ 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



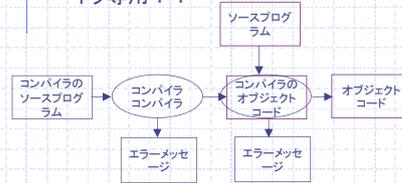
コンパイラ・コンパイラ

- ◆ 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！！

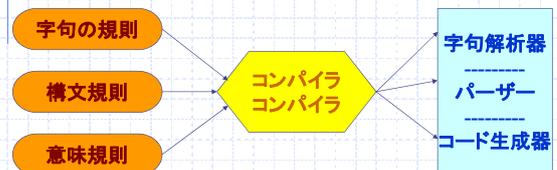


コンパイルのフェーズ

◆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. }

```

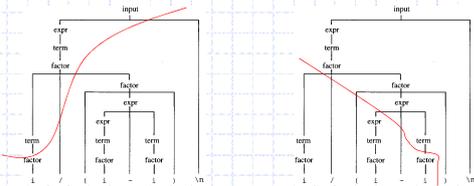
```

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. }

```

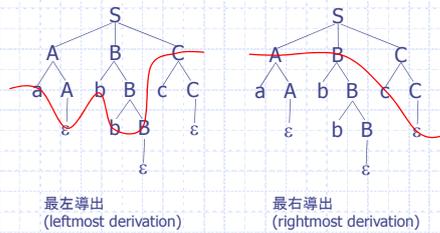

LRとLL

- ◆ 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 $P = \{S \rightarrow ABC, A \rightarrow aA, A \rightarrow \epsilon, B \rightarrow bB, B \rightarrow \epsilon, C \rightarrow cC, C \rightarrow \epsilon\}$.
- ◆ 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
 $S \Rightarrow ABC \Rightarrow aABC \Rightarrow aABcC \Rightarrow abBcC \Rightarrow abBc \Rightarrow abbBc \Rightarrow abbc$
- ◆ 最も左の変数を展開すると
 $S \Rightarrow ABC \Rightarrow aABC \Rightarrow aBC \Rightarrow abBC \Rightarrow abbBC \Rightarrow abbC \Rightarrow abbcC \Rightarrow abbc$
- ◆ 最も右の変数を展開すると
 $S \Rightarrow ABC \Rightarrow ABcC \Rightarrow AbcC \Rightarrow AbBc \Rightarrow Abbc \Rightarrow aAbbc \Rightarrow abbc$
- ◆ 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)