第11章「コンパイラ」

- コンパイラの役割と動作
- 字句解析
- 正規表現
- 構文解析
- BNFと構文解析木
- 最適化

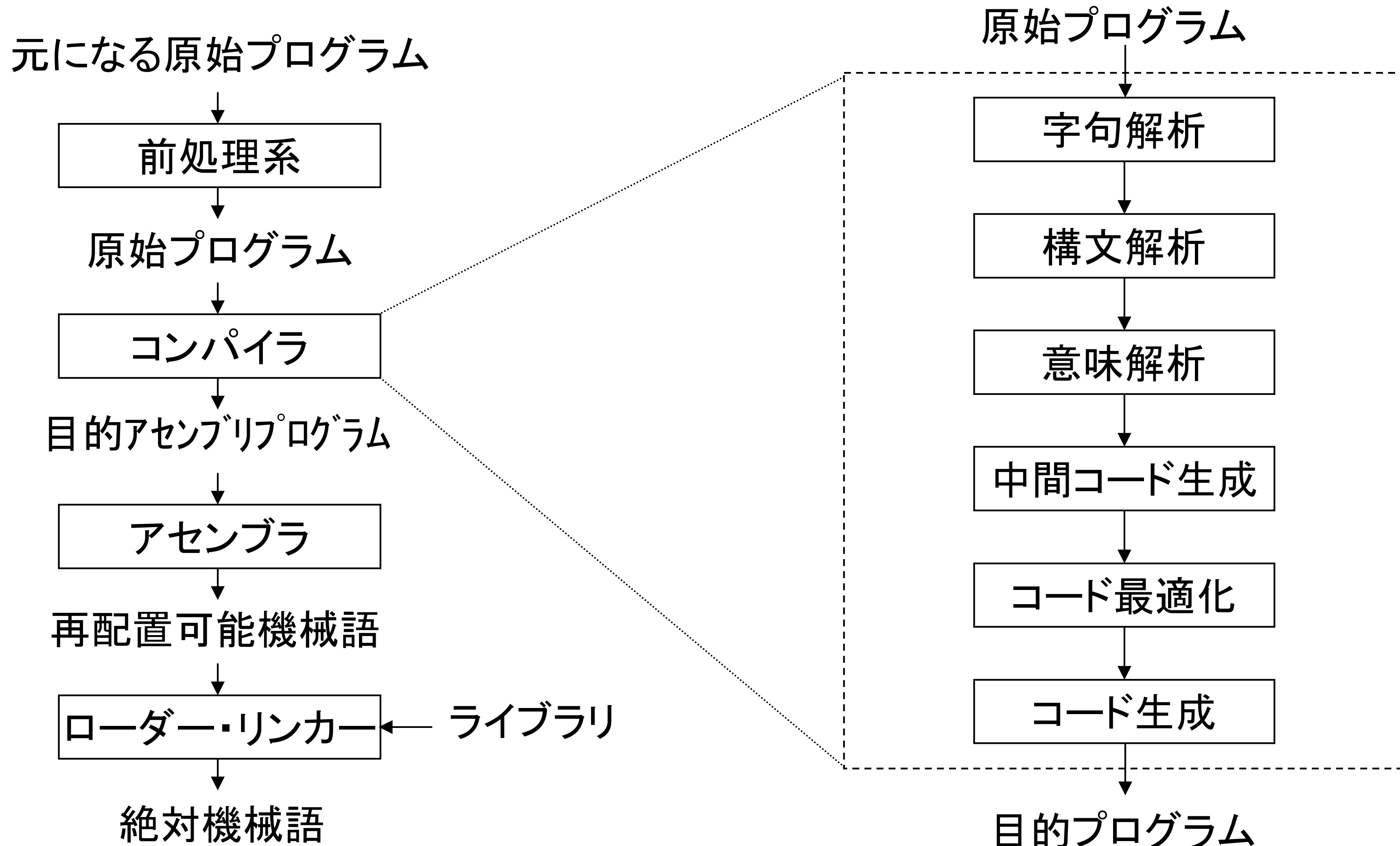
コンパイラとは

- 高水準言語で書かれたプログラムを低水準言語のプログラムに翻訳するプログラム

- 高水準言語の必要性
  - プログラムを簡単に：機械語によるプログラムは複雑
  - プログラム（ソースコード）の再利用を可能に
  - 移植性の向上：CPUごとに機械語は異なる．プログラム言語の仕様が同じならば，同じプログラムが違う計算機の上で動く

コンパイラの位置付けと動作

- プログラム作成から実行まで
- コンパイラの動作



字句解析(lexical analyzer)

- 原始プログラムを字句(トークン, token)と呼ばれる単位に分解する
  - 原始プログラム＝単なる文字列, テキスト

例1)　`area = base*height/2;`

識別子 area
=
識別子 base
*
識別子 height
/
定数 2
;

字句解析(続き)

例2)
```
if (area < 1) {
    base = base * 2;
}
```

予約語 if	識別子 base
(	=
識別子 area	識別子 base
<	*
定数 1	定数 2
)	;
{	}

正規表現(regular expression)

- 文字列の集合を一つの文字列で表現する方法
- 定義の例）文字集合 $S = \{s_1, s_2, \ldots, s_n\}$ に対し，次を定義
  - ε（空文字列）は正規表現である
  - s_i（S の任意要素）は正規表現である
  - R_1, R_2 が正規表現のとき，$R_1 | R_2$，$R_1 \cdot R_2$，R_1* もまた正規表現である．ただし “|” は文字列の並列を，“･” は文字列の連結を，“*”は0回以上の繰返しを表す．演算の順序を表す “(”, “)” は何度使ってもよい（･は省略可）
  - “*” > “･” > “｜”の順に優先
  - 以上の規則によって作られるものだけが正規表現であり，それ以外のものは正規表現でない

正規表現の例

- ε | a | b
  - ε , a, b のいずれか
- a | (ba*)

  例) a, b, ba, baaaaaaa, ･･･
- (a | b)(a | b)(a | b)

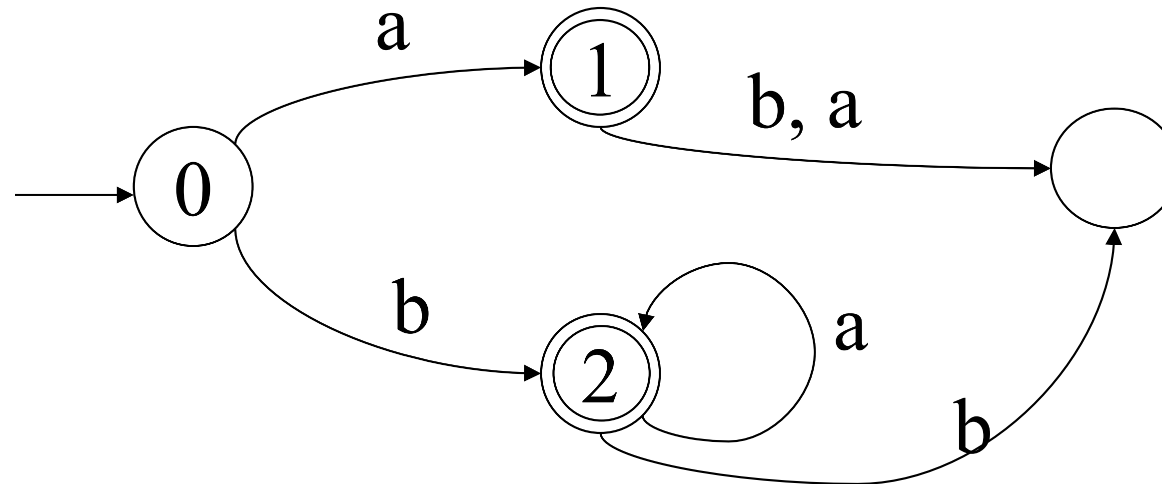
  例) aaa, aab, aba, abb, bbb, ･･･
- (a | b)*

  例) ε , a, ab, abaabaa, b, bbbb, ･･･
- (a | b)*abb

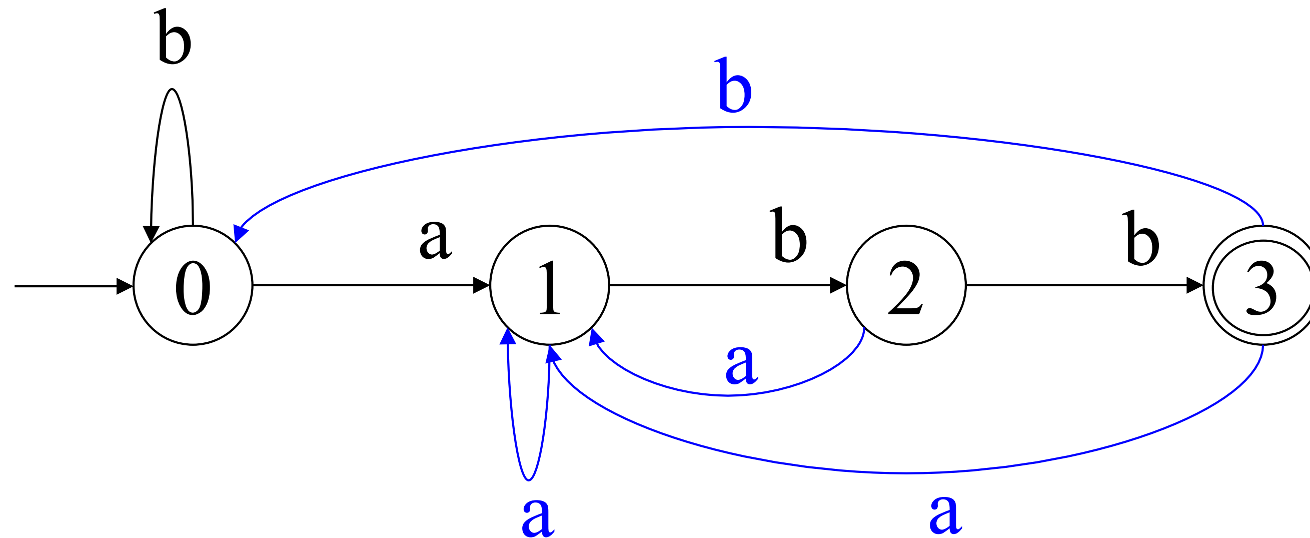
  例) abb, babbbabb, ababb, ･･･

正規表現と順序機械

- a | (ba*)



- (a | b)*abb



正規表現による字句の定義の例

予約語 ＝ for | if | else | return

識別子 ＝ 英字(英字 | 数字)*

定数 ＝ 数字 数字*

演算子 ＝ < | <= | == | >= | >

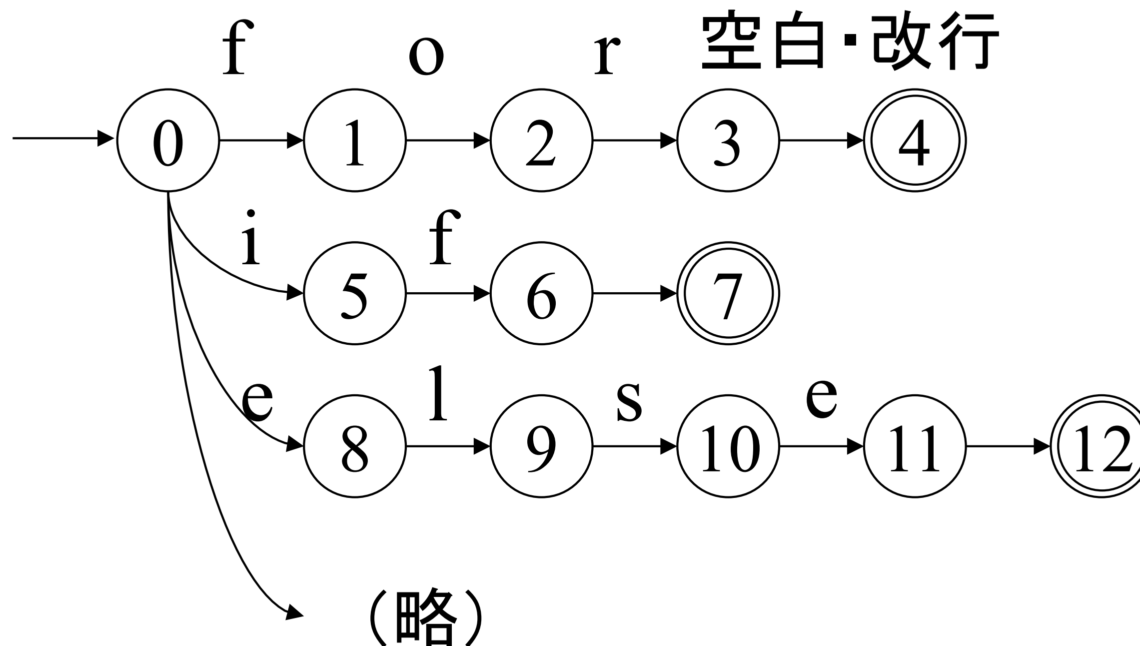
記号 ＝ (|) | { | } | ;

英字 ＝ a | b | ... | z

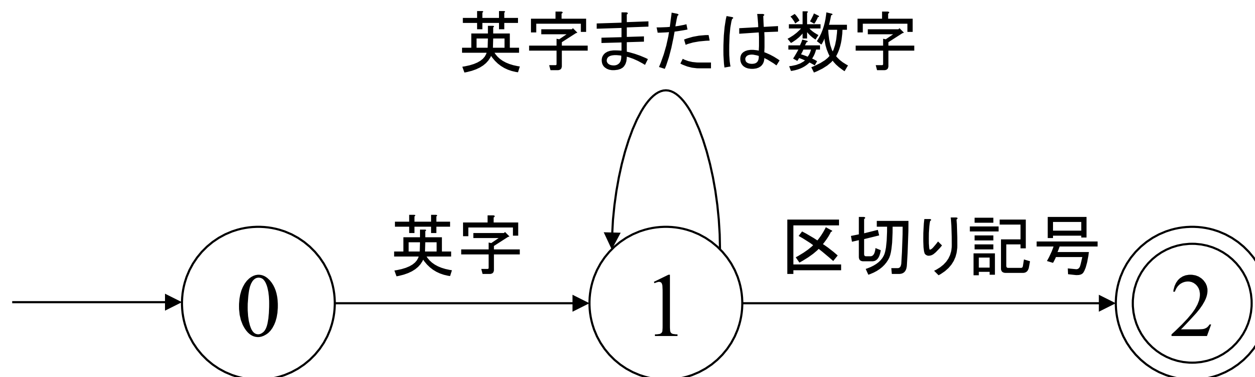
数字 ＝ 0 | 1 | ... | 9

字句の判定

予約語

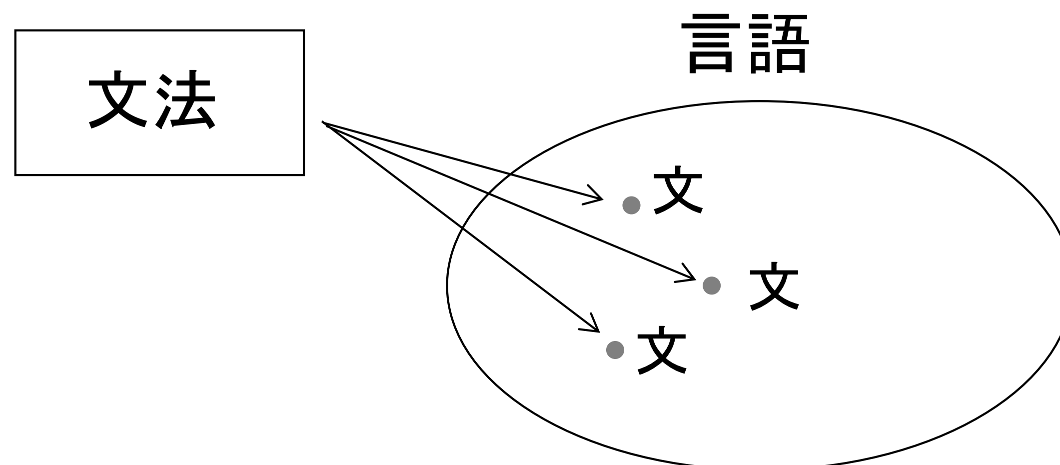


識別子



構文解析

- 字句の列に対して，あらかじめ定める「文法」に照らして構文を認識する
  - 与えられた「文」が文法に則しているかのチェック
  - 「構文解析木」を生成
- 「文法」は可能な文の構造を定義する．導出される文の集合を「言語」と呼ぶ
  - 参考）文脈自由文法（context free grammar）



例：簡略化した日本語の文法

定義する言語の文法

- 文は主部の後に述部がある
- 主部は名詞の後に助詞がある
- 述部は動詞または形容詞である
- 名詞は「私」，「あなた」，「彼」，「彼女」のいずれかである
- 助詞は「は」，「も」，「が」のいずれかである
- 動詞は「走る」，「眠る」のいずれかである
- 形容詞は「可愛い」，「楽しい」のいずれかである

BNF表記

```
文　　　→ 主部　述部
主部　　→ 名詞　助詞
述部　　→ 動詞 | 形容詞
名詞　　→「私」|「あなた」|
          「彼」|「彼女」
助詞　　→「は」|「も」|「が」
動詞　　→「走る」|「眠る」
形容詞　→「可愛い」|「楽しい」
```

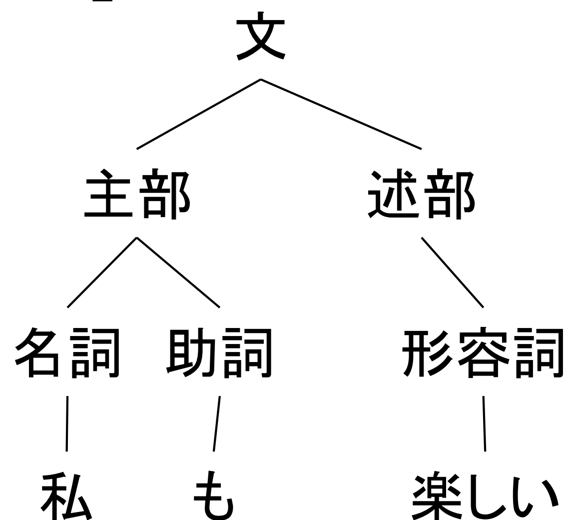
BNF(Backus Naur Form)

- 「a → b」は 生成規則と呼ぶ
- 開始記号
  - 必ず1つだけ存在する（前例では「文」）
- 終端記号（terminal）
  - 「私」や「は」，「走る」など実際に文を構成するもの
- 非終端記号（nonterminal）
  - 文，主部，名詞など実際には文を構成しないもの
- 再帰的な定義も有りうる
  - 例） 文章 → 文 | 文章 文

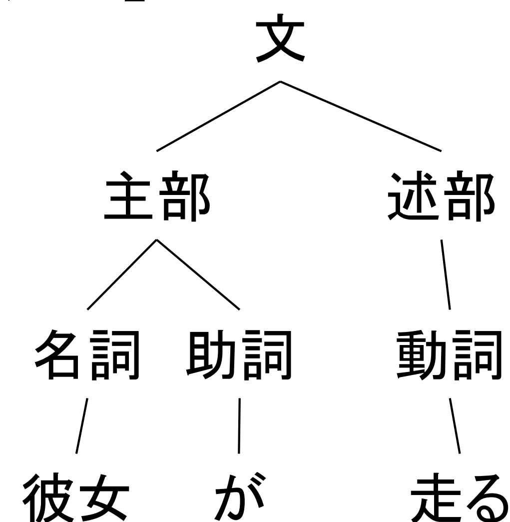
構文解析木

- 個々の文の構造を明らかにしたもの
  - 下のような表現を「木」と呼ぶ
- 文法に従わない字句列は完全な木ができない
  - 「眠る」, 「私も彼女も可愛い」

「私も楽しい」



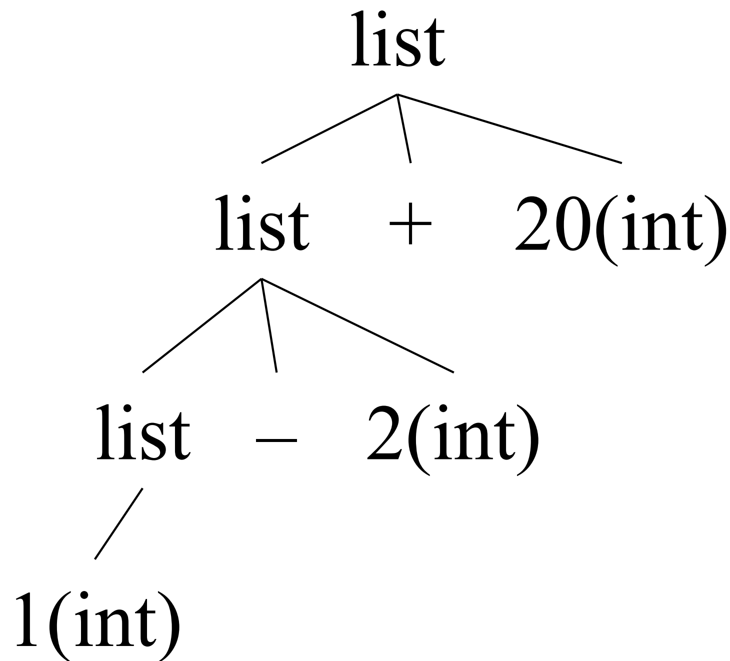
「彼女が走る」



構文解析木(2)

簡単な式の文法： $list \rightarrow list + int \mid list - int \mid int$

例）$1 - 2 + 20$



木の「葉」は終端記号に
「ノード」は非終端記号に
「根」は開始記号に対応

この文法が定める文
ではないもの：

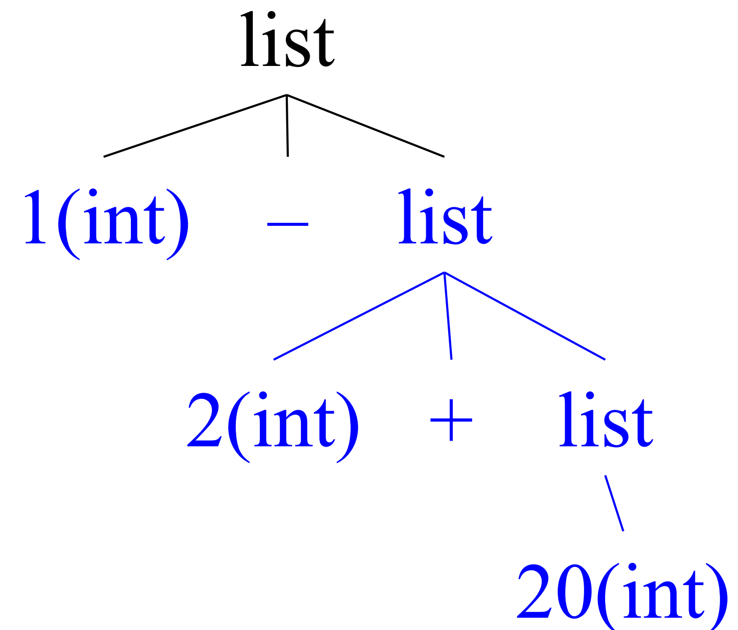
$10 +, - 2$

構文解析木(3)

- 文法の定義しだいで構文解析木は変化する

$$\text{list} \rightarrow \text{int} + \text{list} \mid \text{int} - \text{list} \mid \text{int}$$

1 – 2 + 20



前の定義を左結合，この定義を右結合と言う．言語の意味に応じて使い分ける

例：四則演算は左結合に，べき乗は右結合に

より複雑な式の構文

expr	→	expr + term | expr – term | term
term	→	term * factor | term / factor | factor
factor	→	int | (expr)

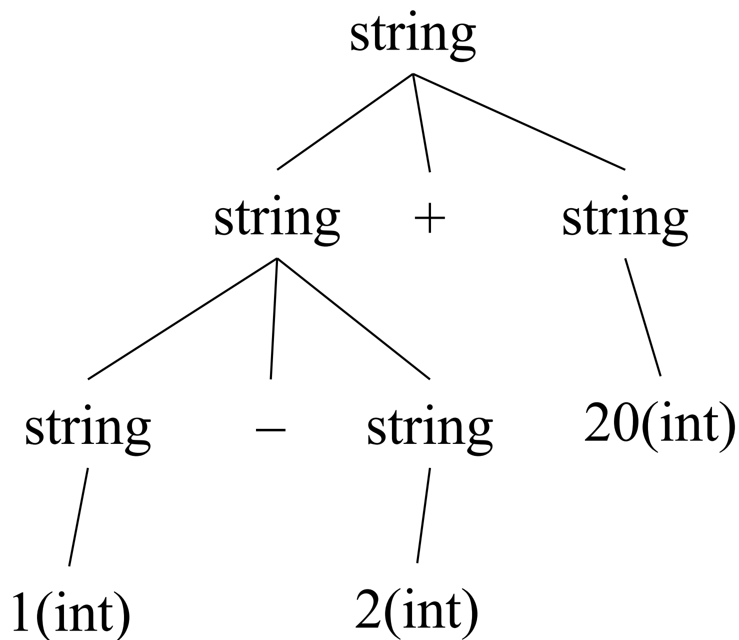
(1 + 2) * 3 – 4 の解析木 →

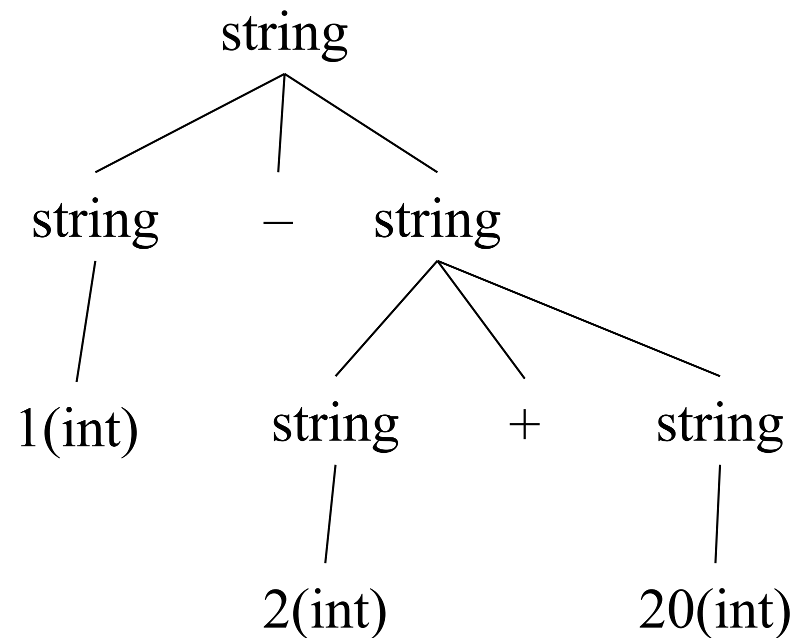
```
expr
├── expr
│   └── term
│       ├── term
│       │   └── factor
│       │       ├── (
│       │       ├── expr
│       │       │   ├── expr
│       │       │   │   └── term
│       │       │   │       └── factor
│       │       │   │           └── 1(int)
│       │       │   ├── +
│       │       │   └── term
│       │       │       └── factor
│       │       │           └── 2(int)
│       │       └── )
│       ├── *
│       └── factor
│           └── 3(int)
├── –
└── term
    └── factor
        └── 4(int)
```

あいまいな文法

・ 曖昧な文法 ＝ 解析木が2つ以上できる文法

例）文法：string → string + string | string – string | integer
に対する 1– 2+20 の解析木（⇔「簡単な式の文法」）





中間コード生成

- 構文解析の結果を用いて，機械語とは直接関係ない中間コードを作成する
- アセンブリ言語と同じ性質をもつ三番地コード（演算子を一つしか許さない）で表現される

$$x := y \; op \; z$$

  - x, y, z は変数名や定数やコンパイラの一時変数
  - *op* は算術演算子や論理演算子など
  - その他，制御の流れを変えるコード（if, goto など）

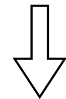
最適化

- （中間コードを）目的に応じてより最適な形に変換すること
- 最適化の目的
  - 実行速度
  - プログラムサイズ（機械語）

最適化の例

1) 無駄を省く

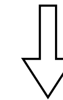
```
    if A > B goto L2
    goto L3
L2:
```



```
    if A <= B goto L3
```

2) 速度向上

```
A := B + C + D
E := B + C + F
```



```
temp1 := B + C
A := temp1 + D
E := temp1 + F
```

3) ループの最適化：

ループ内に同じ結果を生み出す演算があれば
ループの直前に移す

```
area = base * height / 2;
```

↓

字句解析部

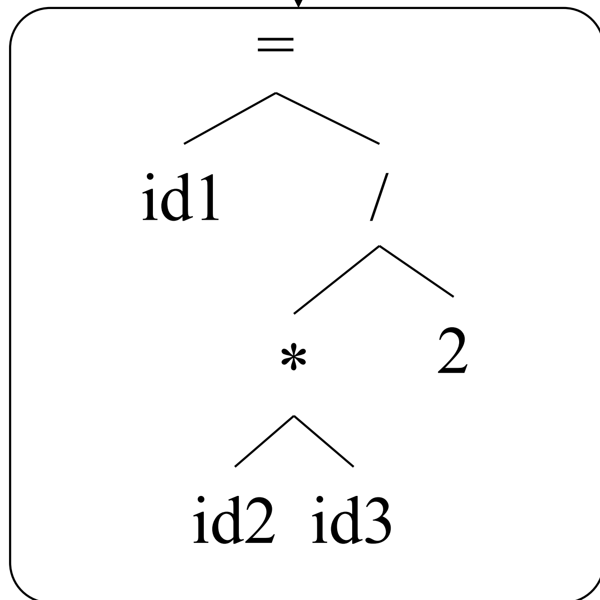
↓

id1 = id2 * id3 / 2

↓

構文解析部

↓



↓

意味解析部

↓

中間コード生成部

↓

temp1 := 2
temp2 := id2 * id3
temp3 := temp2 / temp1
id1 := temp3

↓

最適化部

↓

temp1 := id2 * id3
id1 := temp1 / 2

↓

コード生成

↓

```
LD      A, id2
LD      B, id3
・・・
```

コード生成

- 三番地コード表現された中間コードを機械語に変換
  - レジスタ割り当て：プログラムの高速化のためレジスタをうまく活用する
  - RISC，パイプラインやVLIWのための最適な命令配置を行う