6 構文解析

- ♣ 「構文解析」とはどんなものか
- ♣ 再帰的下降解析法
- ♣ 記号表とその役割

6.1 構文解析とは

1. 構文解析 (☐ , ☐)

   字句解析の結果得られる記号 (☐) の列に対して, その ☐ を調べる

   – 文法的に間違いがあればエラーとする
   – 解析と同時に中間コードや目的コードの生成を行う

2. 構文解析の出力

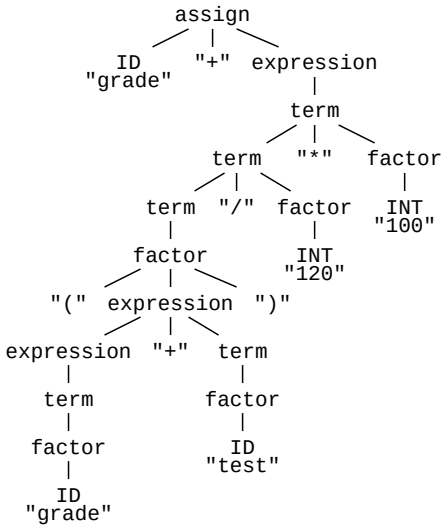
   コンパイラの構成法や目的により, 様々な形式で出力される

```
grade = (test+report)/120*100
```

(a) ☐ (☐ tree)

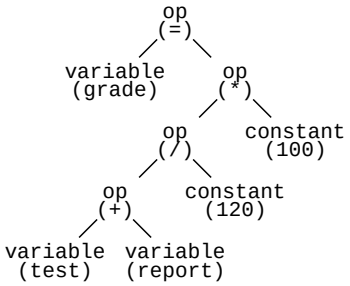
文法構造を示す木

```
assign     ::= ID "=" expression
expression ::= term | expression ("+"|"-") term
term       ::= factor | term ("*"|"/") factor
factor     ::= ID | INT | "(" expression ")"
```



(b) ☐ (☐ tree)

文法構造や意味構造を示す木



☆ 解析木 (☐ tree) とも言う. が, この言葉は導出木を意味することもあり, 結構曖昧.

(c) [ ] 言語 ([ ] language)

i. [ ] 記法 ([ ])

[ ]

ii. [ ] ([ ] , [ ] code)

```
1:   (+,test,report)
2:   [ ]
3:   [ ]
4:   [ ]
```

iii. [ ] ([ ] , [ ] code)

```
1:   (+,test,report,tmp1)
2:   [ ]
3:   [ ]
```

(d) 機械語, アセンブラ

3. 構文解析の条件 (解析の [ ] のため)

記号列の走査は [ ] のみ

k 記号の [ ] を許す (k は通常 [ ] か [ ])

[ ] は許さない

4. 記号表

解析作業の補助に用いる

識別子に関する情報を記録

- [ ] / [ ] 等の区別
- [ ] / [ ] の区別
- [ ]
- [ ]
- メモリー上での [ ]

6.2 再帰的下降解析法

6.2.1 用語

次のような文法

```
文 ::= 主部 述部
主部 ::= 名詞 "は"
述部 ::= 自動詞 | 名詞 "を" 他動詞
名詞 ::= "犬" | "猫" | "ガッキー" | "ポッキー"
自動詞 ::  "走る" | "寝る"
他動詞 ::  "食べる" | "追いかける"
```

において

– [　　] ([　　])

文

– [　　] ([　　])

”犬”, ”猫”, ”ガッキー”, ”ポッキー”, ”は”, ”を”, ”走る”, ”寝る”, ”食べる”, ”追いかける”

– [　　] ([　　])

あるいは [　　] ([　　])

文, 主部, 述部, 名詞, 自動詞, 他動詞

6.2.2 下降解析と上昇解析

下降解析 ([　　] parsing, [　　] parsing)

[　　] から始め, 生成規則の適用 ([　　] : [　　]) を繰り返して, [　　] 記号列を導くことにより, 文法構造を調べる

例) 「ガッキーは ポッキー を 食べる」の解析

文

→ [　　] [　　]

→ [　　] [　　] 述部

→ [　　] "は" 述部

→ "ガッキー" "は" [　　] [　　] [　　]

→ "ガッキー" "は" [　　] "を" 他動詞

→ "ガッキー" "は" "ポッキー" "を" "食べる"

この導出系列から文法構造が分かる

導出系列を求めながら, 解析木, 中間言語, 機械語などを出力する

上昇解析 ([　　] parsing, [　　] parsing)

[　　] から始め, 生成規則の逆の適用 ([　　] : [　　]) を繰り返して, [　　] を導くことにより, 文法構造を調べる

```
"ガッキー" "は" "ポッキー" "を" "食べる"
→ [　　] "は" "ポッキー" "を" "食べる"
→ [　　] "ポッキー" "を" "食べる"
→主部 [　　] "を" "食べる"
→主部 名詞 "を" [　　]
→主部 [　　]
→ [　　]
```

6.2.3 再帰的下降解析法

再帰的下降解析法 ([　　]) とは

- [　　] を利用した比較的簡単な下降解析法
- すべての構文要素に対して, その構文要素を解析する [　　] を一つずつ作る.
- [　　] に対する構文解析ルーチンを呼び出せば, 構文解析は完了する.
- 「どの解析ルーチンを呼び出したか」によって文法構造が分かるので, 解析ルーチンに解析木, 中間言語, 機械語などを生成するコードを組み込んで結果を出力する.

構文解析ルーチンの作り方

1. 文法の右辺に構文要素があれば, [　　] を呼び出す.

   例) 文法規則

   ```
   文 ::= 主部 述部
   ```

   に対する解析するルーチンは,

   ```
   文の解析ルーチン ()
   {
       [　　] の解析ルーチンを呼び出す;
       [　　] の解析ルーチンを呼び出す;
   }
   ```

2. 文法の右辺に終端記号があれば, 解析中の記号列の [　　] がその終端記号と一致しているか調べる. 一致していればその記号を読み飛ばし, 解析を続ける. 一致していなければ [　　] とする.

例) 文法規則

```
主部 ::= 名詞 "は"
```

に対する解析するルーチンは,

```
主部の解析ルーチン ()
{
    [　　] の解析ルーチンを呼び出す;
  if (次の記号が [　　] ) { 次の記号を読み込む;} else { 文法エラー;}
}
```

3. 文法の右辺に [　　] (|) や [　　] (∗) があれば, 記号の先読み (更に先の記号を見ること) により, どちらを選択するか, 繰返しを続行するか終了するかを決定する.

例) 文法規則

```
述部 ::= 自動詞 | 名詞 "を" 他動詞
```

に対する解析するルーチンは,

```
述部の解析ルーチン ()
{
  if (次の記号が [　　] と判断できる) {
    自動詞の解析ルーチンを呼び出す;
  }
  else if (次の記号が [　　] と判断できる) {
    名詞の解析ルーチンを呼び出す;
    if (次の記号が [　　] ) { 次の記号を読み込む;} else { 文法エラー;}
     [　　] の解析ルーチンを呼び出す;
  }
  else { 文法エラー;}
}
```

☆ このような解析を行うには, 文法規則の選択が k 記号の先読み (通常 k は 1 か 2) により行えることが必要になる. このような文法を [　　] 文法という.

6.2.4 mini-C の構文解析の例

1. 「while 文」の解析

文法規則 (BNF) は

```
while 文 ::= "while" "(" 式 ")" 文
```

なので, 解析ルーチンは

```
1:  while 文の解析ルーチン ()
2:  {
3:      if (次の記号が [          ] ) { 次の記号を読む;} else { 文法エラー;}
4:      if (次の記号が [      ] ) { 次の記号を読む;} else { 文法エラー;}
5:          [    ] の解析ルーチンを呼び出す;
6:      if (次の記号が [      ] ) { 次の記号を読む;} else { 文法エラー;}
7:          [    ] の解析ルーチンを呼び出す;
8:  }
```

とすればよい

この文に対するコードは

```
1:  L1 式の
:       コード
2:          [          ]
3:       文の
:       コード
4:          [        ]
5:  L2 …
```

である. 式の解析ルーチンは, 式を [   ] するとともに [   ] を生成するように作ってあるものとする. すると, 式の解析を呼び出した直後に [   ] を生成すれば, 2: までのコード生成ができる. 同様に, 文の解析ルーチンを呼び出すと, [   ] が生成されるので, その直後に [   ] を生成すれば while 文のコード生成が完了する.

```
 1:  while 文の解析ルーチン () /* 文法解析＋コード生成*/
 2:  {
 3:      if (次の記号が "while") { 次の記号を読む;} else { 文法エラー;}
 4:      if (次の記号が "(") { 次の記号を読む;} else { 文法エラー;}
 5:      式の解析ルーチンを呼び出す;
 6:          [                        ]
 7:      if (次の記号が ")") { 次の記号を読む;} else { 文法エラー;}
 8:      文の解析ルーチンを呼び出す;
 9:          [                        ]
10:  }
```

ただし, ”BZ $L2$” を生成する時点では $L2$ の番地が分かっていないので, 工夫が必要 (詳細は演習で).

2. 「文」の解析

BNF は

```
文 ::= ";"
     | "{" 文* "}"
     | if 文
     | while 文
     | return 文
     | 関数呼出し ";"
     | 代入文
```

なので, 解析ルーチンは

```
文の解析ルーチン ()
{
  if      (次の記号が ";") { 次の記号を読む;}
  else if (次の記号が [      ] ) {
    次の記号を読む
    while (次の記号が [      ] でない) {
        [    ] の解析ルーチンを呼び出す;
    }
    次の記号を読む;
  }
  else if (次の記号が "if") { [          ] の解析ルーチンを呼び出す;}
  else if (次の記号が "while") { [              ] の解析ルーチンを呼び出す;}
  else if (次の記号が "return") { [                ] の解析ルーチンを呼び出す;}
  else if (次の記号が 識別子) {
    if (識別子が変数) { [          ] の解析ルーチンを呼び出す;}
    else {
        [                ] の解析ルーチンを呼び出す;
      if (次の記号が ";") 次の記号を読む; else 文法エラー;
    }
  }
  else if (次の記号が "*") { [          ] の解析ルーチンを呼び出す;}
  else { 文法エラー;}
}
```

とすればよい

「文の解析ルーチン」の中から「文の解析ルーチン」自身が呼び出されている (先頭の記号が "{" の場合). このように, 文法中に [   ] があると再帰呼出しが発生する.

3. 「プログラム」全体の解析

元の BNF

```
プログラム ::= ( 関数宣言 | 変数宣言 ";" )*
変数宣言 ::= 型 "*"* ID ( "[" INT "]" )*
関数宣言 ::= 型 "*"* ID "("
            (ε | 変数宣言 ( "," 変数宣言 )* ) ")"
            関数本体
```

では, 1 記号先読みでは「関数宣言」と「変数宣言」のどちらで解析していいのかわからず, プログラムが書けない.

そこで, 文法を次のように変更すると, プログラムが書けるようになる.

```
プログラム ::= ( 宣言頭部 ( 関数宣言尾部 | 変数宣言尾部 ";" ) )*
宣言頭部 ::= 型 "*"* ID
変数宣言尾部 ::= ( "[" INT "]" )*
関数宣言尾部 ::= "(" (ε | 変数宣言 ( "," 変数宣言 )* ) ")"
                関数本体
```

☆ 実際にはこのように構文解析手法に応じて文法を修正することが必要になる.

再帰的下降解析で処理できるように変更した BNF は, 演習の付録参照.

6.3 記号表

記号表の必要性

- 同じ「識別子」でも, それが [　　] か [　　] か, [　　] 変数か [　　] 変数かによって文法や意味が異なることがあり, これらを区別できることが必要
- コードを生成する際, 変数や関数の [　　] などを知る必要がある.
- 同じ識別子が二度宣言されたり, 異なる目的に使われている場合には, これを検出して [　　] と判定する必要がある.

例) mini-c の記号表

- global 記号表と local 記号表がある

  global 記号表… [　　] 変数と [　　] を記録

  local 記号表… [　　] 変数 (および [　　]) を記録

  文の解析中に識別子が現れれば, まず [　　] 記号表を検索し, 次に [　　] 記号表を検索する. (同じ名前の local 変数と global 変数が同時に存在するとき, 関数中ではその識別子は [　　] 変数の方を意味する)

- 記号表の詳細

  各識別子に対して, 次の情報を記録する (括弧内はデバッグ表示する時の文字)

  role … 関数のとき `itab_role_FUNC` (`f`), 変数のとき `itab_role_VAR` (`v`)

  cls … global 変数のとき `itab_cls_GLOBAL` (`g`), local 変数のとき `itab_cls_LOCAL` (`l`), 引数のとき `itab_cls_ARG` (`a`),

  basetype … 識別子が int, int*, int**, ··· と宣言されているとき `itab_basetype_INT` (`i`) とする. char, char*, char**, ··· と宣言されているときには `itab_basetype_CHAR` (`c`) とする.

ptrlevel … 宣言の型に "*" がいくつついているか

argc … 関数の場合にはその引数の数. 変数の場合には配列の次元数 (通常の変数の場合には 0)

aref … 配列変数のとき, 配列表へのインデックス

address … 変数の場合にはその番地. 関数の場合には先頭番地.

size … 変数の場合には主記憶において占めるワード数. 関数の場合にはコードの長さ

- 配列表

  変数が配列変数の場合, その各次元の情報を記録する

  記録する情報はその次元の最大値 (max) と, 要素サイズ (elementsize)

  配列変数の d 次元の情報は, 配列表の (aref の値)+d の位置に登録される

  例えば, 宣言が `int a[3][10][8]` であれば, 表は次のようになる.

	d	max[d]	elementsize[d]
	…	…	…
a の aref →	+0	3	80
	+1	10	8
	+2	8	1
	…	…	…

  すなわち, a[i][j][k] の個々の要素は整数なのでサイズは 1. (注: 実際の c 言語では 2 (バイト) や 4 (バイト) だったりするが, mini-c ではすべてサイズ 1 として扱う.) 従って, elementsize[2]=1 となる

  a[i][j] の各要素は「整数 8 個の配列」と考えることができる. elementsize[1]=max[2] × elementsize[2]=8 となる.

  同様に, a[i] の各要素は「整数 8 個の配列が 10 個集まったもの」と見ることができる. 従って, elementsize[0]=max[1] × elementsize[1]=80 となる.

  配列全体のサイズは size=max[0] × elementsize[0] で 240 となる.

例) プログラム

```
 1:  int a;
 2:  char *b;
 3:  int **c;
 4:  char d[15];
 5:
 6:  int e[4][25];
 7:  char *f[3][7][10];
 8:
 9:  int main()
10:  {
11:  }
12:
13:  int sub1(int p, char *q, int **r, int b)
14:  {
15:     char x; int *y; int z[10][30];
16:  }
```

で, `sub1` を解析している際 (15 行目の解析が終った頃) の記号表の内容は下記のとおり.

global 記号表

番号	識別子	role	cls	basetype	ptr level	argc	aref	add ress	size
0	main	f	g	i	0	0	-1	5	main のコード長
1	a	v	g	i	0	0	-1	0	1
2	b	v	g	c	1	0	-1	1	1
3	c	v	g	i	2	0	-1	2	1
4	d	v	g	c	0	1	0	3	15
5	e	v	g	i	0	2	1	18	100
6	f	v	g	c	1	3	3	118	210
7	sub1	f	g	i	0	4	-1	5	sub1 のコード長

global 配列表

インデックス	max	elementsize
0	15	1
1	4	25
2	25	1
3	3	70
4	7	10
5	10	1

local 記号表

番号	識別子	role	cls	basetype	ptr level	argc	aref	add ress	size
0	p	v	a	i	0	0	-1	0	1
1	q	v	a	c	1	0	-1	1	1
2	r	v	a	i	2	0	-1	2	1
3	b	v	a	i	0	0	-1	3	1
4	x	v	l	c	0	0	-1	4	1
5	y	v	l	i	1	0	-1	5	1
6	z	v	l	i	0	2	0	6	300

local 配列表

インデックス	max	elementsize
0	10	30
1	30	1

Nagisa ISHIURA