(ver. 1.13 笹倉)

抽象構文木 (AST) を用いた中間表現

1　抽象構文木 (AST) とは

抽象構文木 (Abstract Syntax Tree: AST) は，構文木のうちコード生成に必要のない部分を削除したものである．AST の作り方には「こうでなければならない」という規則はない．ただ，コンパイラを実装する上で必要な情報はすべて含んでいる必要がある．

ここでは，情報工学実験 C のコンパイラ作成のための基本言語仕様に合わせた AST の作り方の例を示す．実際に実験で自分のプログラムを作成する時にはこれと異なる作り方をしても全く構わない．また言語仕様を拡張した場合にはそれに合わせた AST にする必要がある．ここで示すものはあくまで例である．

2　基本言語仕様

まずここでは例として情報工学実験で使う基本言語仕様を示す．情報工学実験で作成するコンパイラが入力とする言語の要件は以下のものである．

- 手続き型言語であること
- 代入文、条件分岐文、ループ文があること
- 整数が扱えること
- 四則演算および括弧を含む算術式の計算ができること
- 配列が扱えること

図 1 に示すのが情報工学実験で使う基本言語仕様である．上記の要件を満たしているが，ただ配列の添え字は数字だけが可能となっている．この言語で現実のプログラムを書くにはかなりの制約があり書けないあるいはとても書きにくい問題がたくさんある．実際情報工学実験の最終課題 4 以降を行おうとするとこの基本言語仕様だけでは難しい．最終課題 4 以降を行おうとする場合は配列の添え字に変数も使えるようにする（あるいは算術式も使えるようにする）ことが必要である．またループ文としてこの基本言語仕様では while 文だけがある．while 文以外のループ文例えば for 文を実装するよう拡張するのもよいだろう．比較演算子はこの基本言語仕様では 3 種類しかない．この演算子の種類を増やし，必要なら比較演算子同士の優先順位を実装するのもよい演習である．この言語仕様には関数という概念がない．しかし情報工学実験の最終課題 6(発展課題) を行おうとすると関数が書けなければならない．関数が使えるようにするのことは発展課題とする．

3　情報工学実験用の AST の構造

AST は構文木から必要のないものを省いて作成する．構文木は入力プログラムを文法にしたがって木にしたものだが，それは文法の一つ一つの規則が部分木となり構成される．したがってまず考

```
<プログラム> ::= <変数宣言部> <文集合>
<変数宣言部> ::= <宣言文> <変数宣言部> | <宣言文>
<宣言文> ::= define <識別子>; | array <識別子> [ <数> ];
<文集合> ::=   <文> <文集合>| <文>
<文> ::= <代入文> | <ループ文> | <条件分岐文>
<代入文> ::= <識別子> = <算術式> | <識別子> [ <数> ]= <算術式>;
<算術式> ::= <算術式> <加減演算子> <項> | <項>
<項> ::= <項> <乗除演算子> <因子> | <因子>
<因子> ::= <変数> | (<算術式>)
<加減演算子> ::= + | -
<乗除演算子> ::= * | /
<変数> ::= <識別子> | <数> | <識別子> [ <数> ]
<ループ文> ::= while (<条件式>) { <文集合> }
<条件分岐文> ::= if (<条件式>) { <文集合> }
              | if (<条件式>) { <文集合> } else { <文集合> }
<条件式> ::= <算術式> <比較演算子> <算術式>
<比較演算子> ::= == | '<' | '>'
<識別子> ::= <英字> <英数字列> | <英字>
<英数字列> ::= <英数字> <英数字列>| <英数字>
<英数字> ::= <英字> | <数字>
<数> ::= <数字> <数> | <数字>
<英字> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
         |A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<数字> ::= 0|1|2|3|4|5|6|7|8|9
```

図 1: 情報工学実験のコンパイラの基本言語仕様

えるべきなのは文法の規則一つ一つがどのような部分木になるか，である．実際のコンパイラのプログラムの中では文法のある規則に合致したらそれに対応する部分木を yacc のアクションで生成するように書けば良い．

この節では文法の一つ一つの規則がどのような部分木になればよいかの一例を示す．

抽象構文木にあらわれない終端記号

処理の順番を明示するために使われる終端記号は抽象構文木には現れない．なぜなら抽象構文木の形で処理の順番がわかるから（わかるように抽象構文木を作るから）である．

基本言語仕様の中で処理の順番を明示するための終端記号に該当するのは括弧とセミコロンである．したがって本実験の基本言語仕様であれば，括弧とセミコロンは抽象構文木には現れない（すなわち括弧とセミコロンに対応する部分木は作らなくてよい）．

こどもがない部分木

yacc の規則での終端記号は構文木の葉となる．つまりこどもがいない．例えば，数，変数，演算子などはこどもがない部分木を構成する．

演算子については，今回の基本言語仕様では < 加減演算子 >，< 剰余演算子 > を導入しているのでそれに忠実に構文木を作るとこどもがない部分木となる．ただ，これらの非終端記号を用いないようにして < 算術式 > や < 項 > の右辺に直接演算子があるというように抽象構文木を作ることもできる．そうするとこれらの演算子は二つのこどもを持つ部分木として作ることができる．どちらも構文木としては正しいので自分たちの好きな方で作成すればよい．

こどもが一つの部分木

基本言語仕様にはないが，単項演算子などがこれに当たる．例えばインクリメント演算子やデクリメント演算子などである．返り値を返す return 文などもこれにあたる．

こどもをひとつしか持たない非終端記号、例えば基本言語仕様の中では < 文 > や < 因子 > などは実際にはコード生成には不要なノードである．文法に忠実にこれらのノードを作ってもよいが、効率を考えればこれらのノードは抽象構文木では省く（つまりこれらの非終端記号のノードは作らずその代わりそのこどものノードをその非終端記号の場所につなぐ）とよい．

こどもが二つの部分木

基本言語仕様にある大多数のものがこれになる．算術式にあらわれる二項演算子，比較演算子はこれである．他にも代入文，複数の文を扱えるようにするために導入した文集合，while 文（条件式と文集合）などがこれにあたる．

こどもが三つの部分木

基本言語仕様の中であれば if-else 文がこれにあたる．条件式，条件が真の場合に実行される文集合，条件が負の場合に実行される文集合，の 3 つのこどもを持つ．

こどもが四つの部分木

基本言語仕様の中にはないが，C 言語にあるような for 文を実現しようとするとこれにあたる．for(A;B;C)D とすると，A, B, C, D がこどもとなる．

演習問題

自分たちのグループで定めた言語仕様で，それぞれの規則について抽象構文木を作成した時にいくつこどもを持つ部分木にするかを検討せよ．

4　プログラムの作り方

4.1　抽象構文木のデータ構造

抽象構文木は木である．例えば，二分木であれば，木のノードはたかだか二つのこどもしか持たないのでそのデータ構造は例えば以下のように決めることができる（参考：ウェブページの抽象構文木–準備–二分探索木）．

```
typedef struct btree_node{
  int value;
  struct btree_node *left;
  struct btree_node *right;
} BtreeNode;
```

この BtreeNode 型では，各ノードは整数型の値をもち，left と right の二つのこどもを持つ．こどもがいない場合は left, right に NULL を入れる．

これをもとに今回作成するコンパイラの抽象構文木のデータ構造を考える．拡張しなければいけないところは以下の 2 点である．

- こどもの数がノードによって違う．多い場合は 3 つあるいは 4 つまでサポートする必要がある．
- ノードの持つ値が整数だけとは限らない．

4.2　こどもの数が不定であることへの対処方法

抽象構文木のノードが持つこどもの数がそれぞれのノードによって異なる時にどのようにしてそれを実現するかについて，以下の三つの方法が考えられる．

1. 一番こどもの数が多い場合に合わせてデータ構造を決める
2. リストを使って動的にこどもの数を変更する
3. 配列を使って動的にこどもの数を変更する

4.2.1 一番こどもの数が多い場合に合わせてデータ構造を決める

今回の基本言語仕様では抽象構文木のノードの最大のこどもの数は 3(if-else 文の場合）である．もし拡張して for 文を入れたら最大のこどもの数は 4 となる．

例えば，以下のようなデータ構造にすれば最大のこどもの数 3 までのノード全てに対応できる．こどもがいないところには NULL を入れておけば良い．

```
typedef struct node{
  int value;  // ここは本当はこのノードで必要なデータを入れる．4.3 節参照
  struct node *child1;
  struct node *child2;
  struct node *child3;
} Node;
```

しかし，このデータ構造は以下の二つの理由からあまりお勧めできない．

- 実際には使われない変数がたくさん存在してプログラムの見通しが悪い．
- 実際には使われない変数がメモリを無駄に使っている．

したがって，できれば次のリストを使う方法もしくは配列を使う方法を使うことをお勧めする [1]．

もしこのデータ構造で木を作る場合，新しいノード p を作成するコードは以下のようになるだろう．この例はこどもが二つのノードを作成する例である．ここで p1 は一つ目のこどもへのポインタ，p2 は二つ目のこどもへのポインタとする．

```
Node *p;
p = (Node *)malloc(sizeof(Node));
p->child1 = p1;
p->child2 = p2;
p->child3 = NULL;
```

4.2.2 リストを使って動的にこどもの数を変更する

リスト構造とは，複数のデータを一列に並べたデータ構造のことで，通常 C 言語ではポインタを用いて実現する．図 2 はポインタを使ったリストのイメージ図である．リスト構造の一つ一つの要素は，自分自身の「データを入れる部分」と「次の要素へのポインタ」からなる．最後の要素の「次の要素へのポインタ」には通常 Null を入れる．

以下はこのリスト構造を使った抽象構文木の構造体の例である．ここでは自分の先頭のこどもへのポインタを child に，自分の兄弟のノード（つまり同じ親を持つノード）へのポインタを brother に入れている．この構造体で実現される木のイメージ図を図 3 に示す．

```
typedef struct node{
  int value;  // ここは本当はこのノードで必要なデータを入れる．4.3 節参照
  struct node *child;
  struct node *brother;
} Node;
```

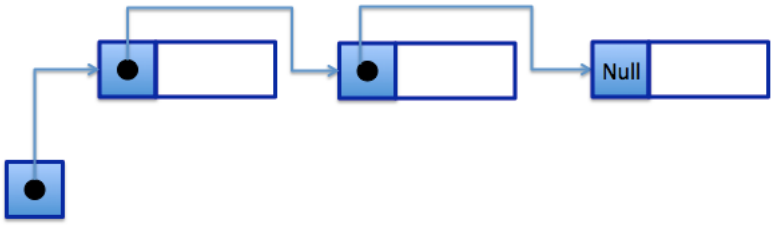

図 2: リスト構造のイメージ

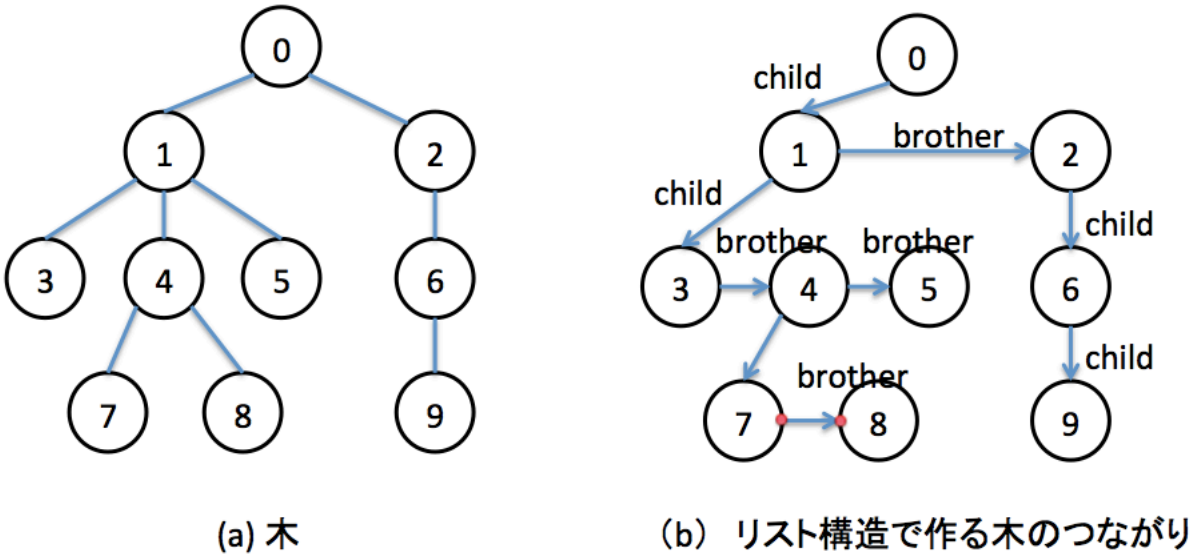

図 3: リスト構造で作った木でのノードのつながり

このデータ構造は次のようなメリットがある．

- 不要な変数がなく，プログラムが比較的わかりやすくなる
- ノードの追加・削除が容易

デメリットは以下である．

- メモリを多く必要とする
- ノードの探索に時間がかかる

デメリットはあるものの，プログラムが比較的書きやすいので，今回の実験のコンパイラの抽象構文木にはリスト構造を用いるのをお勧めする．

もしこのデータ構造で木を作る場合，新しいノード p を作成するコードは以下のようになるだろう．この例はこどもが二つのノードを作成する例である．ここで p1 は一つ目のこどもへのポインタ，p2 は二つ目のこどもへのポインタとする．このノードの兄弟ノードはとりあえずないものとして作成する．

[1] しかし，考え方としてわかりやすいので本実験ではこのデータ構造を使うという選択もあり．

```
Node *p;
p = (Node *)malloc(sizeof(Node));
p->child = p1;
p1->brother = p2; // p->child->brother = p2 でもよい
p->brother = NULL;
```

4.2.3 配列を使って動的にこどもの数を変更する

前項のリストを使う方法と同じ考え方だが，ポインタを使ったリストではなく，こどもを配列を使って実現する方法を紹介する．ただしこどもの数は不定なので，あらかじめ決まった大きさの配列を取るのではなく，配列へのポインタを宣言しておいて，実際に配列に入るべき要素の個数がわかってから配列を malloc で確保する．

以下はこどもの管理に配列を使って木を作成するためのデータ構造の例である．このデータ構造を使う時にはこどもの数を記録する変数を作っておく必要がある．

```
typedef struct node{
  int value;  // ここは本当はこのノードで必要なデータを入れる．4.3 節参照
  int number; // こどもの数
  struct node *child;
} Node;
```

このデータ構造を用いて新しいノードを作成する時のコードは例えば以下のようになるだろう．この例はこどもが二つのノードを作成する例である．ここで p1 は一つ目のこどもへのポインタ，p2 は二つ目のこどもへのポインタとする．こどもの数が 2 であるということは与えられているとする．

```
Node *p;
p = (Node *)malloc(sizeof(Node));
p->number = 2;
p->child = malloc(sizeof(Node*) * p->number)
p->child[0] = p1;
p->child[1] = p2;
```

このデータ構造のメリットは

- メモリの使用効率がよい
- 実行が早い

ことである．一方でデメリットは

- プログラムが読みにくくなる
- デバッグがしにくくなる

ことである．

一般的には読みやすくデバッグしやすいプログラムの方がよいのでリストを用いた方法の方がお勧めである．しかし，メモリを少しでも節約したい場合や，少しでも早いプログラムが求められるような場合はこの方法のデータ構造を用いると良い．

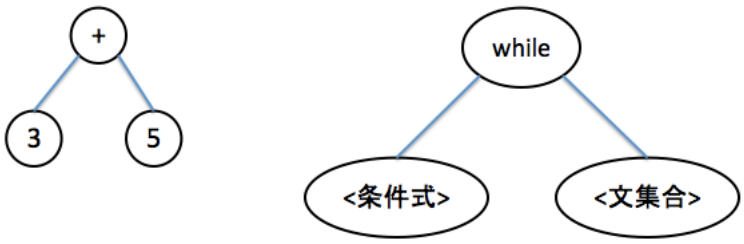

図 4:

4.3 ノードが持つべき情報

ここまでで木の形を作ることはできるはずである．次に抽象構文木のノードが持つべき情報は何かを考える．これまでの例では int value; としていた部分である．

例えば 3 + 5 といった足し算の文と while (< 条件文 >) { < 文集合 > } の二つの例を考える．これはどちらもこどもを二つ持つ図 4 のような部分木になるであろう．

この二つの部分木は形はまったく同じである．言い換えればこの左側が 3+5 を表す部分木で右側が while 文を表す部分木であるということが区別できるような情報をノード自身が持つ必要がある．具体的に言うと，このノードは何を表すノードかを示すノートタイプという変数を作りそれに例えば+ならば PLUS, while ならば WHILE というようなノードタイプを示す値を入れておけば良い．

ノードタイプとしてどのようなものを用意するかに決まりはないが，例えば，抽象構文木に現れるすべてのノードごとにノードタイプをつけるとわかりやすい．列挙型で以下のように定義すればよい[2] [3]．

```
typedef enum {
  IDENT_AST, // 変数のノード型
  NUM_AST,   // 整数のノード型
  ASSIGN_AST,  // 代入文の=のノード型
  PLUS_AST, // 加算演算子のノード型
  MINUS_AST, // 減算演算子のノード型
  ...<中略>...
  WHILE_AST,  // while のノード型
  IF_AST,     // if のノード型
  ...<後略>...
} NType;
```

その上で例えばノードの型を

```
typedef struct node{
  Ntype type;
```

[2] グループによっては新しい列挙型を作らなくても lex から yacc へ渡すトークンの種類で代用できる可能性があるので検討してみること．

[3] トークンの種類とは別に定義する場合にはトークン名と同じにならないように例えば_AST をつけるなどの工夫をするとよい．

```
  struct node *child;
  struct node *brother;
} Node;
```

のように定義して，ノードを作成する時に例えば

```
Node *p;
p = (Node *)malloc(sizeof(Node));
p->type = WHILE_AST;
p->child = p1;
p1->brother = p2; // p->child->brother = p2 でもよい
p->brother = NULL;
```

のようにすればよい．この例ではこのノードは while 文のノードとしている．

さらに数値や変数の時には，コードを生成する時にその値が必要となる．したがってノードに数値や変数の情報を加えるのがよい．例えばノードの型を以下のようにする．

```
typedef struct node{
  Ntype type;
  int   ivalue; // 整数の場合の値を入れる
  char* variable; // 変数の場合変数名の文字列を入れる
  struct node *child;
  struct node *brother;
} Node;
```

この例では変数の場合はこの段階では文字列として格納することにしている [4].

演習問題

1. 自分たちのグループで定めた言語仕様で，< 文集合 > ，< 文 > といった非終端記号をどのように扱うかを考えてみよう.（抽象構文木は最終的に正しいコードがでればどのように作っても良いので「こうすべき」というような正解はない.）
2. 必要なノード型の名前を決めよう（名前はなんでもいい）.
3. 配列の場合，例えば a[5] + 1 の文を表す抽象構文木を作るにはノード型をどうすればよいか考えてみよう.
4. ここで例に挙げたノードの型では，ivalue はノードが整数の場合にしか使われず，variable はノードが変数の場合にしか使われない．共用体を使って無駄を省くにはどうしたらよいか考えてみよう.

[4] まだ説明していないが、記号表を作る場合には文字列ではなく記号表を参照するような値を格納するという方法がある．本実験ではとりあえず今は文字列へのポインタにしておいて記号表を作る段階でこの部分を見直すことにする．4.4 節参照.

4.4 変数の取り扱い

変数は，一般的には抽象構文木だけでは十分な情報が扱えないので記号表 (symbol table) を用いて管理する．ただ，この実験の基本言語仕様の場合，関数を扱えず，変数の型が一種類しかないので記号表を作らなくてもコード生成することが可能である．4.3 節に示したノード型は基本言語仕様で記号表を作らない場合を想定したものである．

しかし，関数を導入したり，複数の型の変数を導入するように言語仕様を拡張した場合，記号表を作って変数を管理するのがよい．あるいは，定義されていない変数を使った場合にエラーメッセージを出したいと思えば[5]，記号表を使って定義された変数が何かを管理する必要がある．このような場合は，抽象構文木には変数名の文字列ではなく記号表の中のその変数の変数番号 (もしくはポインタ) を入れるとよい．

記号表には一般的には，変数番号，変数名，型を入れる．最終的にはその変数が割り当てられたメモリ上の番地も入ることになる．変数番号とはその変数にユニークになるようにつけた番号のことである．変数を記号表に入れる時に順次つけていけばよい．関数などがある場合には変数のスコープ情報も必要である．スコープ (ブロック) ごとに記号表を分けるという方法を取るのが普通である．

記号表は字句解析の時に作成するという方法もあるが，本実験では構文解析時に宣言文が出てきたらそのアクション部で記号表にその変数の情報を追加するという方法を推奨する．

記号表のデータ型の例を示す．基本言語仕様では型は一種類という想定なので型情報は入れていない．リストで複数の変数を入れられるようにしているが，リストではなく配列を用いるという実現もできる．配列の方が実行速度が速くなるので大きなプログラムをコンパイルするような場合は配列の方が望ましいが，本実験で作成するコンパイラはそれほど大きなプログラムをコンパイルすることは想定していないのでリストでも配列でもどちらでもよい．

```
typedef symbols{
  int symno; // 変数番号．ユニークになるようにつけること
  char *symbolname; // 変数名
  struct symbols* next; // 次の変数へのリンク
}Symbols;
```

5 yacc ファイルの書き方

抽象構文木は，yacc のアクション部で作成する．手順としては以下のように考えると分かりやすい．

1. それぞれの規則でどのような部分木を作成すればよいかはすでに検討した．それに基づいて各規則のアクション部に抽象構文木を作成する C コードを書く．
2. アクション部で書いたコードで用いる$$がすなわりその規則の左辺のシンボル値である．このシンボルの型を適切に定める．

[5] このようなチェックは構文解析ではできない．

5.1 アクション部の作成例

5.1.1 例：加算演算子の部分のプログラムの作り方

加算演算子の部分を例にとってプログラムの作り方を解説する．
加算演算子の部分の文法は

<算術式> ::= <算術式> <加減演算子> <項>

で，この加算演算子の部分が yacc の規則部で次のように表されているとする．

```
expression : expression PLUS term
```

```
expression : expression PLUS term {$$ = build_node2(PLUS_AST, $1, $3);}
```

ここで，関数 build_node2(NType, Node*, Node*) はこどもを二つ持つノードを作るための関数，PLUS は加算を示すノードタイプである．関数 build_node2(NType, Node*, Node*) の中身は例えば以下のようにすればよい．

```
Node*
build_node2(NType t, Node* p1, Node* p2){
   Node *p;
   if(p = (Node *)malloc(sizeof(Node)) == NULL){
      yyerror("out of memory");
   }
   p->type = t;
   p->child = p1;
   p1->brother = p2; // p->child->brother = p2 でもよい
   p->brother = NULL;

   return p;
}
```

この関数 build_node2(NType, Node*, Node*) は yacc ファイルのユーザ定義部に入れてもいいが，そうするとこの関数を修正するたびに yacc を通さなければならなくなる．よって別の C のソースファイルを作ってそれに入れたほうがよい（参考：ウェブのスライド資料「分割コンパイル」）．

5.1.2 例：数値のノードの作り方

数値のノードが出てくる部分の文法は

<変数> ::= <識別子>| <数> | <識別子> [<数>]

で，この数の部分が yacc の規則部で次のように表されているとする．

```
var : NUMBER
```

ここで NUMBER は数を表すトークンの種類である．数を表す抽象構文木のノードを作成するにはこの規則のアクション部に例えば次のように書けばよい．

```
var : NUMBER { $$ = build_num_node(NUM, $1);}
```

ここで，関数 build_num_node(NType, int) は数に対応したノードを作成する関数，NUM は数を示すノードタイプである．lex ファイルでトークンが数の時にその値を整数値にして yylval に入れていると仮定している．したがって$1 にはこの数の数値が入っているはずである．関数 build_num_node(NType, int) の中身は例えば以下のようにすればよい．

```
Node*
build_num_node(NType t, int n){
   Node *p;
   if(p = (Node *)malloc(sizeof(Node)) == NULL){
      yyerror("out of memory");
   }
   p->type = t;
   p->ivalue = n
   p->child = NULL;
   p->brother = NULL;

   return p;
}
```

5.1.3 例：変数のノードの作り方

いわゆる変数（文法規則上では識別子）のノードが出てくる部分の文法は

```
<変数> ::= <識別子>| <数> | <識別子>  [ <数> ]
```

で，この識別子の部分が yacc の規則部で次のように表されているとする．

```
var : IDENT
```

ここで IDENT は識別子を表すトークンの種類である．識別子を表す抽象構文木のノードを作成するにはこの規則のアクション部に例えば次のように書けばよい．

```
var : IDENT { $$ = build_ident_node(IDENT, yytext);}
```

ここで，関数 build_ident_node(NType, char*) は識別子に対応したノードを作成する関数，IDENT は識別子を示すノードタイプである．ここでは yytext を用いて識別子の具体的な文字列を取得している．yytext は文字列へのポインタで，常に字句解析された最新のトークンを指しているので，もし例のようなコードにする場合には build_indet_node 関数の中で mallloc してメモリを確保し yytext の中身をそこにコピーしておかないと yylex が次の字句を解析したときに中身が変わってしまう．ここで yytext を参照するのではなく，lex ファイルの中で yytext の中身をあらかじめ別の変数にコピーしそのポインタを yylval の値として渡すという方法もある [6]．どのような方法をとるかはそれぞれのグループで話し合って決めてもらいたい．関数 build_ident_node(NType, char*) の中身は例えば以下のようにすればよい．

[6] そもそも lex の段階で記号表を作ってしまって lex からその記号表へのポインタ（もしくは変数番号）を yylval で yacc に渡すという方法もある．

```
Node*
build_ident_node(NType t, char *s){
   Node *p;
   if(p = (Node *)malloc(sizeof(Node)) == NULL){
      yyerror("out of memory");
   }
   p->type = t;
   if(p->variable = (char *)malloc(sizeof(MAXBUF)) == NULL{
      yyerror("out of memory");
   }
   strncpy(p->variable, s, MAXBUF);
   p->child = NULL;
   p->brother = NULL;

   return p;
}
```

これは識別子のノードに変数名の文字列を入れることを想定した場合のコードである．MAXBUF は変数名の最大の長さを決める記号定数で，#define で値が決められているものと想定している．

記号表を使う場合[7] には，例えば

```
Node*
build_ident_node(NType t, char *s){
   Node *p;
   if(p = (Node *)malloc(sizeof(Node)) == NULL){
      yyerror("out of memory");
   }
   p->type = t;
   p->variable = SLookup(s);
   p->child = NULL;
   p->brother = NULL;

   return p;
}
```

などのようになるであろう．ただし，ここで関数 SLookup(char*) は記号表から引数で与えられた文字列の変数の変数番号（もしくはポインタ）を検索してくる関数であるとし，p→variable の型は関数 SLookup(char*) の戻り値と同じ型にしなければならない．

[7]記号表を使うにはどこかで記号表を作っておかなければならない．普通は 1) 字句解析をしながら (すなわち lex で) 作る 2) 構文解析しながら変数の宣言文のところがきたらその変数の宣言を記号表に付け加えることで作る，のどちらかになるであろう．

5.2 シンボルの型の指定

各規則のアクション部のコードの中に$$を使う場合，その型が C のプログラムとして考えたときに正しく合っていなければならない [8].

シンボルの型の指定は yacc の定義部で行う．シンボルがトークンである場合と非終端記号の場合で定義の仕方が違う．

シンボルの型の定義

シンボルの型は YYSTYPE で定義される．デフォルトは int である．lex からシンボル値を受け取るための変数 yylval は YYSTYPE の型で宣言される．

シンボル値の型が一種類でかつ int 以外のものを使いたい場合は#define 文を使って YYSTYPE を設定すれば良い．例：#define YYSTYPE double [9]

シンボル値の型がトークンや非終端記号ことに違う場合は，yacc ファイルの中で%union を使って指定する．%union をつかうと YYSTYPE の型がそこで宣言されたものになる．例えば，シンボル値として整数，文字列へのポインタ，抽象構文木のノード型 Node へのポインタの 3 種類を可能にするときは以下のように yacc ファイルの定義部に書く．

```
%union{
  Node* np;  // 抽象構文木
  int   ival; // 数
  char* sp;  // 変数名
 }
```

これは最終的に C ファイルに変換されるので，抽象構文木のノードの型 Node はこの%union 文より前で定義されていなければならない．Node が%union より前に定義されていない場合は C のコンパイルエラーになる．

トークンの場合

トークンの場合には%token を使ってトークンのシンボル値の型を定義する．以下の例は，シンボル値の型が前述のように定義されていたときに，IDENT というトークンのシンボル値は文字列へのポインタ，NUMBER というトークンのシンボルは整数であることを意味している． これらは yacc ファイルの定義部に書く．

```
%token <sp> IDENT
%token <ival> NUMBER
```

非終端記号の場合

非終端記号の場合には%type を使ってトークンのシンボル値の型を定義する．以下の例は，シンボル値の型が前述のように定義されていたときに，expression term factor という非終端記号のそれぞれのシンボル値は Node へのポインタ，add_op mul_op という非終端記号のそれぞれのシンボルは整数であることを意味している． これらは yacc ファイルの定義部に書く．

```
%type <np> expression term factor
%type <ival> add_op mul_op
```

[8] アクション部に何も書かなかったときは，$$=$1 が設定される．これをデフォルトアクションとも呼ぶ．つまり$$と$1 が同じ型でなければコンパイルエラーが出る．

[9] yacc の演習問題：実数の四則演算参照.

6　抽象構文木を作る演習課題

ここまでの解説を参考にして，自分たちのグループで定義した言語について yacc のアクション部を AST を作るように作成せよ．作成した AST が正しいかどうかを確認するために AST を表示する関数を作成し，構文解析終了後，main 関数から AST を表示する関数 printTree ををを呼ぶこと．printTree を作成するにあたっては二分探索木の資料（解答例）が参考になるだろう．

main 関数は yacc のユーザ定義サブルーチン部ではなく，別の C のソースファイルにするのが望ましい．main 関数の中身は概ね次のようなものになるであろう．きちんと動くプログラムにするためにこれに必要なコードを付け加えること．

```
Node *parse_result = NULL; // 抽象構文木のルートノードを受け取るための変数
int
main(void){
  int result;

  result = yyparse();
  if(!result){
    printTree(parse_result); // 抽象構文木を出力するための関数.
  }
  return 0;
}
```

抽象構文木のルートノードは yacc のスタートシンボルのシンボル値になる（ように作る）ので，それを yacc のアクションでグローバル変数　parse_result に入れておく．このプログラムでは変数 parse_result の宣言は yacc ファイルとは別にした main 関数があるファイルで行っているので yacc のファイルの方では extern 宣言が必要である．

参考文献

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey Ullman, “Compilers: Principles, Techniques, and Tools (2nd Edition)”, Addison Wesley, 2006.