コンパイラ

第7回 制約検査

— 型の検査と表管理 —

http://www.info.kindai.ac.jp/compiler

E館3階E-331 内線5459

takasi-i@info.kindai.ac.jp

コンパイラの構造

- 字句解析系
- 構文解析系
- 制約検査系
- 中間コード生成系
- 最適化系
- 目的コード生成系

制約検査系 (constraint checker)

- 制約検査系
  - 変数の未定義・二重定義・型の不一致などを検査

変数 x は未定義

変数 i は配列ではない

代入の左辺が変数ではない

```
int i, j;
x = 0;
i[10] = 5;
0 = 10;
```

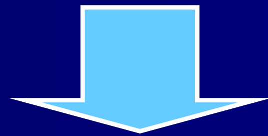
制約検査

- 制約検査
  - 変数の未定義・二重定義
    - 変数が定義されているか
  - 代入の左辺値
    - 代入の左辺に左辺値はあるか
  - 型検査
    - 式の型が要求される型に一致しているか

変数の未定義･二重定義

- 変数の未定義
  - 変数宣言していない変数を使用
- 変数の二重定義
  - 宣言済の変数を再度宣言

共に制約エラー



変数名の管理が必要　　記号表を使用

記号表(symbol table)

- 記号表
  - 名前(変数名, 関数名など)を管理
- 記号表の項目
  - 名前
  - 種類
    - 変数名, 定数名, 手続き名, 関数名, 型名など
  - 型
    - int 型, double 型, array of int 型, pointer 型等
  - 記憶位置
    - 記憶番地 (変数の場合)
    - 実行開始番地 (手続き, 関数の場合)
    - 値そのもの(定数の場合)

記号表

例 : int i; char ch; int a[10];
int max (int, int); void printArray (int[]);
const PI = 3.1416;

名前	種類	型	番地
i	変数	int	0
ch	変数	char	1
a	変数	array of int [10]	2～11
max	関数	int × int → int	1000
printArray	関数	array of int [] → void	2000
PI	定数	double	3.1416

変数表

- 変数表
  - 変数名, 型, 記憶番地等

例 : int i, j=1; char ch; double d; int a[10]; double m[3][5];

名前	型	サイズ	番地	代入	参照
i	int	1	0	未	未
j	int	1	1	済	未
ch	char	1	2	未	未
d	double	2	3〜4	未	未
a	array of int [10]	1*10	5〜14	未	未
m	array of double [3][5]	2*3*5	15〜44	未	未

関数表

- 関数表
  - 関数名, 型(引数, 返り値),実行開始番地等

例 : int max (int, int);
void printArray (int[]);

名前	型	番地
max	int × int → int	1000
printArray	array of int [] → void	2000

記号表の探索

- 名前をキーとして探索
  – 線形探索
  – ハッシュ探索
  – 2分探索

名前 “ans”

名前	種類	型	番地
ans	変数	int	15

線形探索

- 線形探索
  - 上から順に探索

名前 “max”

	ans	変数	int	0
	time	変数	int	1
	mat	変数	array of int [5]	2〜6
	max	関数	int × int → int	1000
-	data	変数	double	7〜8

ハッシュ探索

- ハッシュ探索
  - 名前のハッシュ値で探索

名前 “data”

hash (“data”) = 6

0	-
1	→ PI / 定数 / double / 3.14 / -
2	-
3	→ ans / 変数 / int / 15 / → time / 変数 / double / 25 / -
4	-
5	-
6	→ data / 変数 / String / 30 / -
7	-

2分探索

- 2分探索
  – 2分木上を探索

名前 “val”

各探索方法の長所と短所

探索方法	探索時間	長所	短所
線形探索	遅い $O(n)$	簡単	遅い
ハッシュ探索	速い $O(1)$	最も速い	要ハッシュ関数
2分探索	速い $O(\log n)$	速い	要ソート

原始プログラムが適度にモジュール化されていれば線形探索でも充分に高速

Var クラス, VarTable クラス

Var		# 変数定義部
- type	: Type	# 変数の種類
- name	: String	# 変数の名前
- address	: int	# 変数のDseg上のアドレス
- size	: int	# サイズ
Var (type : Type, name : String, addr : int)		# コンストラクタ
getType ()	: Type	# 変数の種類を返す
getName ()	: String	# 変数の名前を返す
getAddress ()	: int	# 変数のアドレスを返す
getSize ()	: int	# 変数のサイズを返す

VarTable		# 変数表定義部
- varList	: ArrayList<Var>	# 変数表
- nextAddress	: int	# 次の変数のアドレス
VarTable ()		# コンストラクタ
- getVar (name : String)	: Var	# 変数を返す
exist (name : String)	: boolean	# 変数の存在判定
registerNewVariable (type : Type, name : String, size, int)	: boolean	# 変数表に要素追加
getAddress (name : String)	: int	# アドレスを返す
getType (name : String)	: Type	# 種類を返す
checkType (name : String, type : Type)	: boolean	# 型の一致判定
getSize (name : String)	: int	# 変数のサイズを返す
size ()	: int	# 表のサイズを返す
removeTail (index : int)	: void	# 表の末尾を削除する

変数表への挿入

- 変数表への挿入

```
/** @ return 変数 name を登録できたか？ */
boolean registerNewVariable
    (Type type, String name, int size)
```

例：int i, j;

型は INT

スカラ変数のサイズは 1

```
varTable. registerNewVariable (Type.INT, “i”, 1);
varTable. registerNewVariable (Type.INT, “j”, 1);
```

例：int a[5], b[] = {1, 2, 3};

型は ARRAYOFINT

配列のサイズ

```
registerNewVariable (Type.ARRAYOFINT, “a”, 5);
registerNewVariable (Type.ARRAYOFINT, “b”, 3);
```

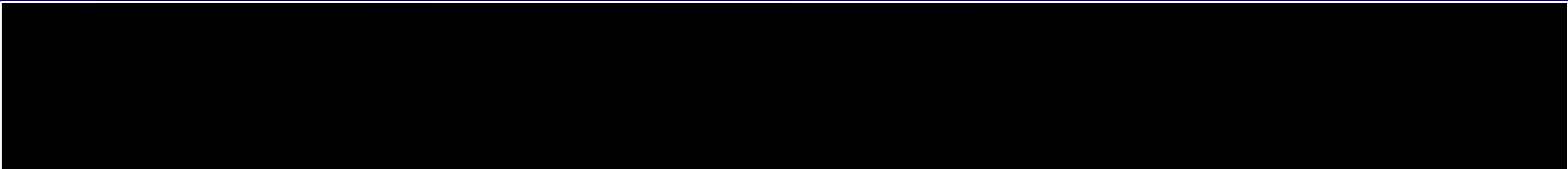
VarTable.java

nextAddress 0 初期値 0

varList : ArrayList<Var>

type	name	address	size

空の ArrayList



VarTable.java

size を
足す

nextAddress 1

varList : ArrayList<Var>

type	name	address	size
INT	i	0	1

Var クラスのオブジェクトを生成

```
registerNewVariable (Type.INT, “i”, 1);
```

VarTable.java

nextAddress 51

varList : ArrayList<Var>

type	name	address	size
INT	i	0	
ARRAYOFINT	a	1	50

```
registerNewVariable (Type.INT, "i", 1);
registerNewVariable (Type.ARRAYOFINT, "a", 50);
```

変数表への登録判定

- 変数表への登録判定は
  VarTable.exist (String) を使用

```
/** @ return 変数 name は登録済か？ */
boolean exist (String name)
```

例 : 変数 x は登録済か？

```
varTable.exist ("x")
```

変数の型判定

- 変数の型判定は

VarTable.checkType (String, Type) を使用

```
/** @ return 変数 name の型が type か？ */
boolean checkType (String name, Type type)
```

例 : 変数 i は int 型か？

```
varTable.checkType (“i”, Type.INT)
```

変数の番地

- 登録された変数の番地を得るには

  VarTable.getAddress (String) を使用

```
/** @ return 変数 name の番地 */
int getAddress (String name)
```

登録されていない変数の場合は返り値は -1

例 : 変数 i の番地

```
varTable.getAddress (“i”)
```

VarTable.java

nextAddress 152

varList : ArrayList<Var>

type	name	address	size
INT	i	0	1
ARRAYOFINT	a	1	50
ARRAYOFINT	b	51	100
INT	n	151	1

exist (“n”) ⇒ true

checkType (“i”,Type.INT) ⇒ true

checkType (“x”,Type.ARRAYOFINT) ⇒ false

VarTable.java

nextAddress 152

varList : ArrayList<Var>

type	name	address	size
INT	i	0	1
ARRAYOFINT	a	1	50
ARRAYOFINT	b	51	100
INT	n	151	1

getAddress (“i”) $\Rightarrow$ 0

getAddress (“b”) $\Rightarrow$ 51

getAddress (“x”) $\Rightarrow$ -1

変数表のサイズ

- 変数表のサイズ(登録されている変数の個数)

  VarTable.size () を使用

```
/** @return 変数表のサイズ */
int size ()
```

VarTable.java

nextAddress 202

varList : ArrayList<Var>

type	name	address	size
INT	i	0	1
ARRAYOFINT	a	1	50
ARRAYOFINT	b	51	100
INT	n	151	1
ARRAYOFINT	c	152	50

```
size()
```

$\Rightarrow$ 5

変数表からの削除

- 変数表の末尾に登録された変数を削除するには VarTable.removeTail (int) を使用

```
/** index 番目以降の変数を削除 */
void removeTail (int index)
```

登録されている変数の個数以上の
値を指定した場合は何もしない

例 : 5番目以降の変数を削除

```
varTable.removeTail (5)
```

VarTable.java

nextAddress 151

varList : ArrayList<Var>

	type	name	address	size
0	INT	i	0	1
1	ARRAYOFINT	a	1	50
2	ARRAYOFINT	b	51	100
3	INT	n	151	1
4	ARRAYOFINT	c	152	50

removeTail (3)

VarTable.java

nextAddress 151

varList : ArrayList<Var>

	type	name	address	size
0	INT	i	0	1
1	ARRAYOFINT	a	1	50
2	ARRAYOFINT	b	51	100

removeTail (3)

VarTable.java

nextAddress 151

varList : ArrayList<Var>

	type	name	address	size
0	INT	i	0	1
1	ARRAYOFINT	a	1	50
2	ARRAYOFINT	b	51	100

```
removeTail (10)
```
⇒ 何もしない

```
removeTail (-1)
```
⇒ 何もしない

変数の型

- マクロ構文から変数の型を判定

<Decl> ::= “int” NAME [“[” <Const> “]”] “;”

int i ; の場合 : INT 型

int a [10] ; の場合 : ARRAYOFINT 型

“[” <Const> “]” の有無で型を決定

制約検査プログラム

- 変数宣言部 (スカラ変数の場合)

```
void parseVarDecl () {
    if (token == “int”) token = nextToken();
        else syntaxError();
    if (token == 名前) {
        String name = token.strValue の値; // 変数名を得る
        token = nextToken();
    } else syntaxError();
    if (exist (name))                    // すでに登録済かをチェック
        syntaxError (“二重登録です”); // 二重登録は制約エラー
    registerNewVariable (INT, name, 1);   // 変数表に登録
      :
```

```
     :
 if (token == 名前) {
    String name =  token.strValue の値;   // 変数名を得る
    token = nextToken();
 } else syntaxError();
 if (exist (name))  syntaxError ();     // 二重登録は制約エラー
 if (token == “[”) {                     // 配列の場合
    token = nextToken();
    if (token == 整数) {
       int size =  token.intValue の値;   // 整数値を得る
       token = nextToken();
    } else syntaxError();
    if (token == “]”) token = nextToken(); else syntaxError();
    (ARRAYOFINT, name, size) で変数表に登録;
} else {                                 // スカラ変数の場合
    (INT, name, 1) で変数表に登録;
      :
```

初期値あり配列

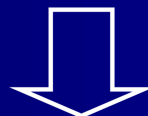
- 変数表への登録にはサイズが必要

```
int a[] = { 10, 20, 30 } ;
```

サイズ未定

“}” まで読めば
サイズ確定

“}” まで読んだ時点で変数表に登録する



初期値の個数をカウントしておく

```
if (token == “int”) token = nextToken();
    else syntaxError();
if (token == NAME) token = nextToken();
    else syntaxError();
if (token == “[”) { // 配列の場合
    token = nextToken();
    if (token == INTEGER) { // 初期値無しの配列
        “[” INTEGER “]” “;” の解析
        変数表に登録
    } else if (token == “]”) {   // 初期値有りの配列
        “[” “]” “=” “{” <Const> { “,” <Const> } “}” “;” の解析
        変数表に登録
    }
} else { // スカラ変数の場合
```

<Const> の個数をカウント

<Unsigned>部の制約検査

- マクロ構文から変数の型を判定

<Unsigned> ::= NAME
| NAME “[” <Exp> “]”
| INTEGER | CHARACTER | …

“[” <Exp> “]” の有無で型を決定

“[” <Exp> “]” 無し ⇒ INT型以外はエラー
“[” <Exp> “]” 有り ⇒ ARRAYOFINT型以外はエラー

制約検査プログラム

- <Unsigned> 部 (スカラ変数の場合)

```
void parseUnsigned () {
 if (token == 名前) {
      String name = token.strValue の値;   // 変数名を得る
      if (!exist (name))                    // 登録済かをチェック
         syntaxError (“未定義です”);        // 未定義ならエラー
      token = nextToken();
      if (!checkType (name, INT))   // 登録された型をチェック
         syntaxError (“型が不一致です”);  // int 型以外はエラー
     }
  } else if (token == 整数) {
                 :
```

```
boolean parseUnsigned () {
  if (token == 名前) {
    String name = token.strValue の値;    // 変数名を得る
    if (!exist (name)) syntaxError (“未定義です”);
    token = nextToken();
    if (token == “[” ) {                  // 配列の場合
      if (name の型が ARRAYOFINT 以外)
        syntaxError (“型が不一致です”);
      “[” <Exp> “]” の処理
    } else {                              // スカラ変数の場合
      if (name の型が INT 以外)
        syntaxError (“型が不一致です”);
    }
  } else if (token == 整数) {
```

変数の型

- int 型以外の型がある場合

<Decl> ::= “int” NAME [“[” <Const> “]”] “;”
| “double” NAME [“[” <Const> “]”] “;”
| “char” NAME [“[” <Const> “]”] “;”
| “String” NAME [“[” <Const> “]”] “;”

型	“[” <Const> “]”	
	無し	有り
int	INT	ARRAYOFINT
double	DOUBLE	ARRAYOFDOUBLE
char	CHAR	ARRAYOFCHAR
String	STRING	ARRAYOFSTRING

スコープルール(scope rule)

- スコープルール
  - 名前の有効範囲

```
if (a == 0) {
    int x;
      :
}
for (int i=0; i<10; ++i) {
      :
}
```

int 型変数 x は
この内部のみで有効

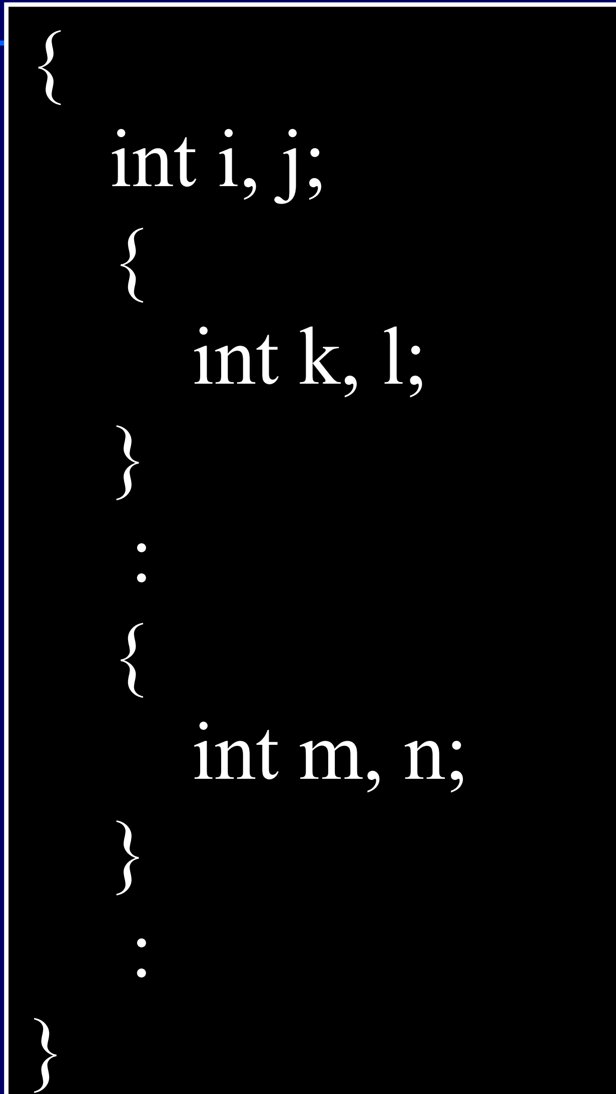
int 型変数 i は
この内部のみで有効

有効範囲ごとに記号表を作成する

スコープルール

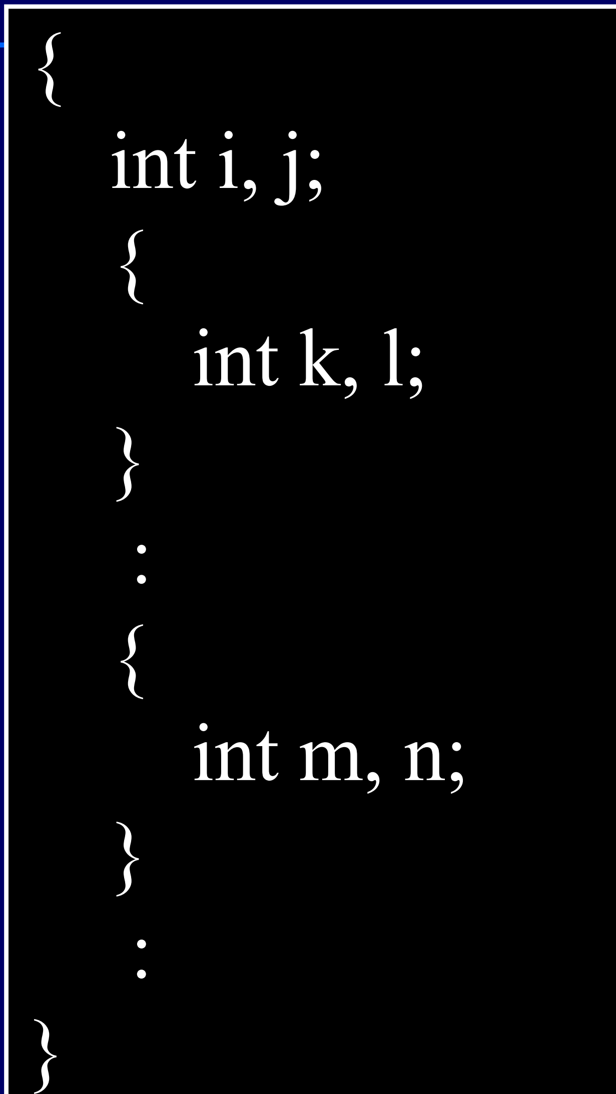
- 記号表の動的管理
  - ブロックに入る → 新しい記号表を作成
  - ブロックから出る → 最新の記号表を削除
- 名前の参照
  - 最も新しい記号表から順に検索

記号表の動的管理



i	変数	int	0
j	変数	int	1

記号表の動的管理



i	変数	int	0
j	変数	int	1

k	変数	int	2
l	変数	int	3

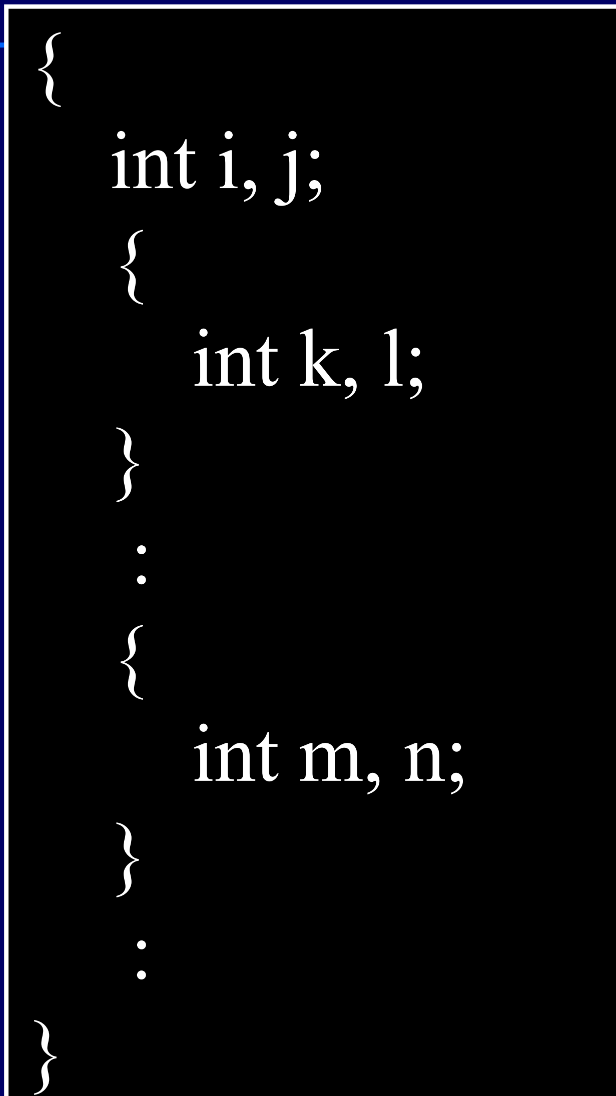
記号表の動的管理

```
{
    int i, j;
    {
        int k, l;
    }
     :
    {
        int m, n;
    }
     :
}
```

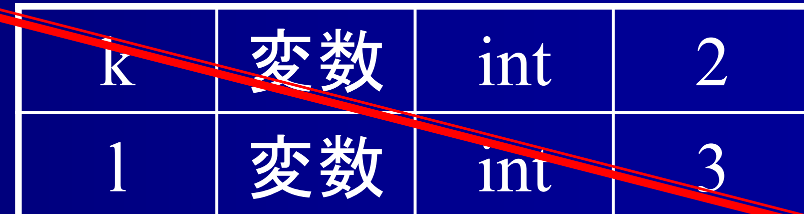
i	変数	int	0
j	変数	int	1

k	変数	int	2
l	変数	int	3

記号表の動的管理



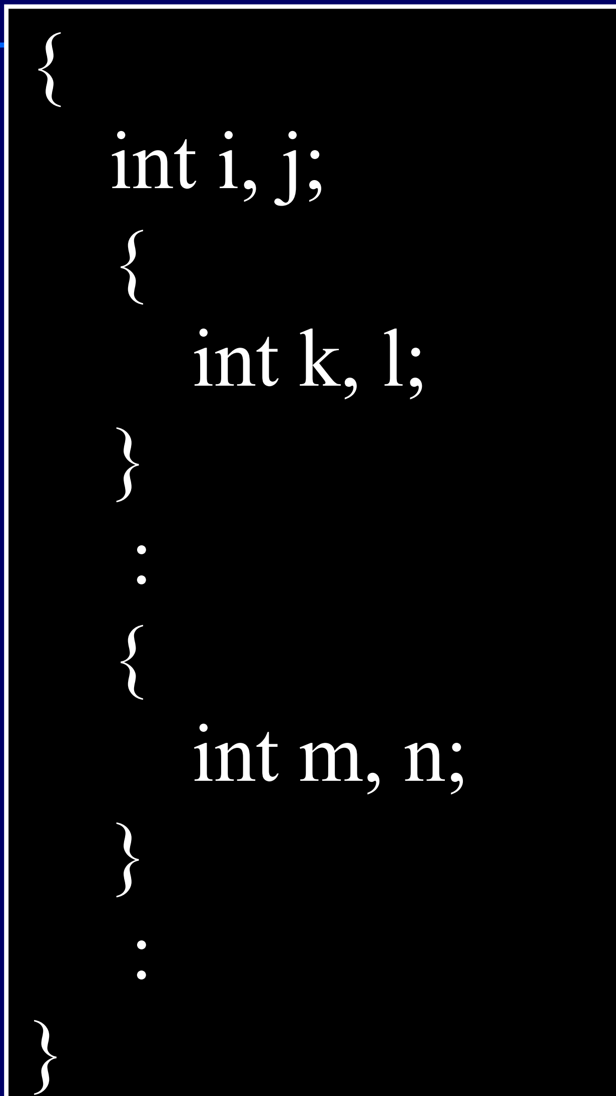
i	変数	int	0
j	変数	int	1



m	変数	int	2
n	変数	int	3

k と m, l と n で
共通の領域を使用

記号表の動的管理

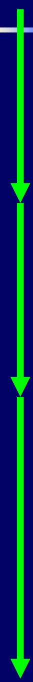


i	変数	int	0
j	変数	int	1

k	変数	int	2
l	変数	int	3

m	変数	int	2
n	変数	int	3

名前の参照



```
{
    int i, j, k;
    {
        int i, j;
        {
            int i
        }
        :
    }
    :
}
```

i	変数	int	0
j	変数	int	1
k	変数	int	2

i	変数	int	3
j	変数	int	4

i	変数	int	5

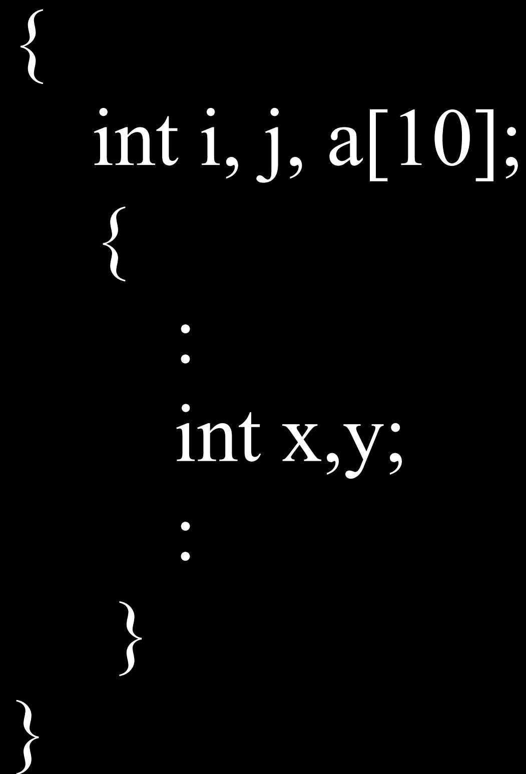
変数 i の参照

変数 j の参照

変数 k の参照

最新の表から順に参照

VarTable の管理(拡張課題)

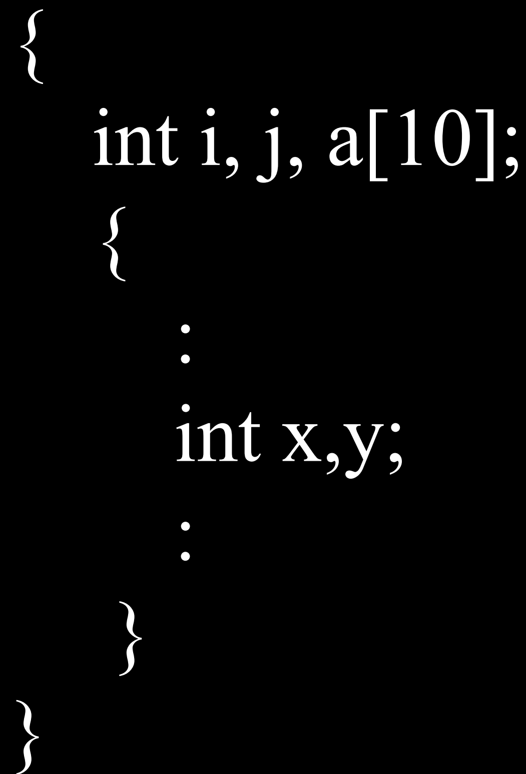


type	address	name	size
INT	0	i	1
INT	1	j	1
ARRAYOFINT	2	a	10

```
int tableSize = varTable.size();
```

ブロック開始時点の変数表のサイズを記憶

VarTable の管理(拡張課題)



type	address	name	size
INT	0	i	1
INT	1	j	1
ARRAYOFINT	2	a	10
INT	12	x	1
INT	13	y	1

```
varTable.removeTail (tableSize);
```

ブロック終了時に変数表の末尾を削除

VarTable の管理(拡張課題)

```
for (int i=0; i<n; ++i ) {

    :

}
```

int 型変数 i は
この内部のみで有効

for 文開始時に変数表のサイズを記憶
for 文終了時に変数表の末尾を削除

左辺値(left value, locator value)

- 左辺値(left value, locator value)
  - 代入の左辺として認められる値

    i = 10;   a[5] += b[10];   x = y = z;

- 右辺値(right value)
  - 代入の左辺とならない値

    10 = j;   a + b = c;   i ++ = ++ j;

右辺値が代入の左辺に来ると制約エラー

構文規則と左辺値

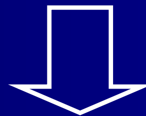
- 構文規則
  - <Expression> ::= <Exp> [“=” <Expression>]
  - <Exp> ::= <Exp> “+” <Term> | <Term>
  - <Term> ::= <Term> “*” <Factor> | <Factor>
  - <Factor> ::= <Unsigned> | “-” <Factor>
  - <Unsigned> ::= [“++”] NAME
| [“++”] NAME “[” <Exp> “]”
| INTEGER | CHARACTER
| “(” <Exp> “)”

構文規則上では代入の左辺の制約無し
⇒構文解析とは別に制約検査が必要

左辺値の判定

- 左辺値の判定
  - 非終端記号解析時に左辺値の有無を返す

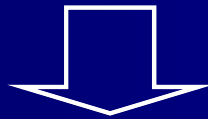
```
void parse<A>() {
 <A> のマクロ構文と合致するか判定
}
```



```
boolean parse<A>() {
 <A> のマクロ構文と合致するか判定
 return <A>が左辺値を持つか？
}
```

左辺値の判定

- <Unsigned> の判定
  - 左辺値を持つのは 変数単独, 配列単独の場合のみ



スカラ変数, 配列なら true,
それ以外なら false を返す

左辺値の判定

```
boolean parseUnsigned () {
    switch (token) {
      case 名前 :                          // 変数の場合
          token = nextToken();
          return true;                      // 左辺値あり
      case 整数 :                          // 整数の場合
          token = nextToken();
          return false;                     // 左辺値無し
      case "(" :                           // "(" <Exp> ")" の場合
          token = nextToken();
          parseExp();
          if (token == ")") token = nextToken();
              else syntaxError();
          return false;                     // 左辺値無し
}
```

左辺値の判定

何らかの演算を行うと左辺値が無くなる

- <Factor> の場合
  - <Factor> ::= <Unsigned> | “-” <Factor>

<Factor> → “-” <Factor> の場合

“-” 演算を行った ⇒ 左辺値を持たない

<Factor> → <Unsigned> の場合

<Unsigned> が左辺値を持つなら持つ

左辺値の判定

```
boolean parseFactor () {
    if (token == “-”) {              // “-” <Factor> の場合
        token = nextToken();
        parseFactor();
        return false;                  // 左辺値無し
    } else if (token ∈ First (<Unsigned>)) {
                                       // <Unsigned> の場合
        boolean hasLeftValue = parseUnsigned();
                    // <Unsigned> の左辺値の有無をコピー
        retuen hasLeftValue;
    } else syntaxError();
}
```

左辺値の判定

- <Term> の場合
  - <Term> ::= <Factor> { “*” <Factor> }

<Term> → <Factor> “*” <Factor> の場合

“*” 演算を行った ⇒ 左辺値を持たない

<Term> → <Factor> の場合

<Factor> が左辺値を持つなら持つ

左辺値の判定

```
boolean parseTerm () {
    if (token ∈ First (<Factor>)) {
       boolean hasLeftValue = parseFactor();
                         // <Factor> の左辺値の有無をコピー
    } else syntaxError();
    while (token == “*”) {
        token = nextToken();
        if (token ∈ First (<Factor>)) {
            parseFactor();
        } else syntaxError();
        hasLeftValue = false;
                            // 掛け算があるので左辺値無しに
    }
    return hasLeftValue;
}
```

左辺値の判定

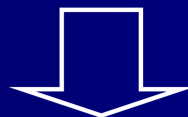
- <Expression> の場合
  - <Expression> ::= <Exp> [“=” <Expression>]

<Expression> → <Exp> の場合

<Exp>は左辺値を持たなくてもよい

<Expression> → <Exp> “=” <Expression> の場合

<Exp> は左辺値が必要



左辺値がなければ制約エラー

左辺値の判定

```
void parseExpression () {
   if (token ∈ First (<Exp>)) {
      boolean hasLeftValue = parseExp();
                        // <Exp> の左辺値の有無をコピー
      else syntaxError();
   if (token == "=") {
      if (!hasLeftValue)
         syntaxError ("左辺値がありません");
                              // 左辺値が無ければエラー
      token = nextToken();
      if (token ∈ First (<Expression>))
         parseExpression();
         else syntaxError();
   }
}
```

返り値を用いない左辺値判定

- フィールド変数 hasLeftValue を使用

```
boolean hasLeftValue;  // 直前の式が左辺値を持つか
```

```
void parseUnsigned () {
    switch (token) {
      case 名前 :                    // 変数の場合
          token = nextToken();
          hasLeftValue = true;              // 左辺値あり
      case 整数 :                    // 整数の場合
          token = nextToken();
          hasLeftValue =  false;            // 左辺値無し
                :
}
```

返り値を用いない左辺値判定

```
void parseExpression () {
    if (token ∈ First (<Exp>)) {
        parseExp(); // この中でフィールド変数の値が設定
    } else syntaxError();
    if (token == “=”) {
        if (!hasLeftValue)  // フィールド変数で判定
            syntaxError(); // 左辺値が無ければエラー
        token = nextToken();
        if (token ∈ First (<Expression>)) {
            parseExpression();
        } else syntaxError();
    }
}
```

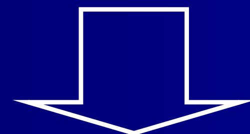
多重定義(overloading)

- 多重定義(overloading)
  - 一つの記号が異なる意味を持つ

例：“-”

<E> ::= <T> “-” <T>　：2項演算子

<F> ::= “-” <F> | <U> : 単項演算子



“-” がどちらの意味で使用されているか
コンパイル時に判別が必要

多重定義の判別

- 構文解析時に判別可能な例
  - <E> ::= <T> “-” <T>
  - <F> ::= “-” <F> | <U>
- 構文解析時に判別不可能な例
  - <T> ::= <F> “*” <F>

  int 型 * int 型 → int 型
  double 型 * double 型 → double 型

  <F> の型により演算結果の型が変わる

数値の表現

- 機械語レベルでの数値の表現
  - 型により表現が異なる

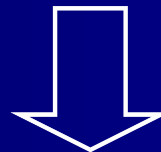
メモリ

例 : int 型 15

15

double 型 3.1416
= 31416 * 10^{-4}

31416
-4



機械語レベルでは型により異なる処理が必要

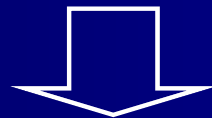
式の型

- 演算によって得られる型
  - 被演算子の型に依存

例: <F> “*” <F>

int 型 * int 型 → int 型
double 型 * double 型 → double 型

どちらも掛算だが計算機にとっては
int 型と double 型は異なる処理が必要



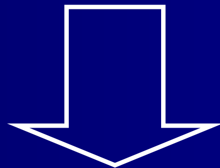
被演算子の型検査が必要

型検査

- 型検査
  - 式の型と要求されている型を比較

例 : <T> ::= <F> “%” <F>

“%” (剰余演算子)の被演算子は整数同士のみ定義



2つの <F> が整数型でなければエラー

型制約規則

- 型制約規則
  - 式が取ることができる型
    - 式に対する型検査で使用

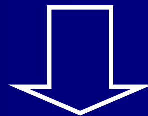
型制約規則の例

生成規則	型制約規則
<F> ::= INTEGER	Type (<F>) = Type.INT
<F> ::= CHARACTER	Type (<F>) = Type.CHAR
<F> ::= NAME	Type (<F>) = varTable.getType (NAME)
<T> ::= <F>$_1$ “*” <F>$_2$	if (Type(<F>$_1$) == Type.INT && Type(<F>$_2$) == Type.INT) Type (<T>) = Type.INT else Type (<T>) = Type.DOUBLE

式の型判定

- 式の型判定
  - 非終端記号解析時に式の型を返す

```
void parse<A>() {
 <A> のマクロ構文と合致するか判定
}
```



```
Type parse<A>() {
 <A> のマクロ構文と合致するか判定
 return <A>の型
}
```

式の型判定

- <Unsigned> の型
  - int 型定数の場合 : int 型
  - double 型定数の場合: double 型
  - char 型定数の場合 : char型
  - 変数の場合 : 変数表に登録された型
  - “(” <Exp> “)” の場合 : <Exp> の型

:

式の型判定

- <Unsigned> の型 (定数の場合)

```
Type parseUnsigned () {
   switch (token) {
     case 整数 :                        // 整数の場合
         token = nextToken();
         return Type.INT;               // int 型
     case 実数 :                        // 実数の場合
         token = nextToken();
         return Type.DOUBLE;            // double 型
     case 文字 :                        // 文字の場合
         token = nextToken();
         return Type.CHAR;              // char 型
              :
```

式の型判定

- <Unsigned> の型 (変数, “(” <Exp> “)” の場合)

```
case 名前 :                    // 変数の場合
    String name = token.strValue の値; // 変数名を得る
    token = nextToken();
    return varTable.getType (name);
                               // 変数表に登録されている型
case “(” :                     // “(” <Exp> “)” の場合
    token = nextToken();
    Type type = parseExp(); // <Exp> の型をコピー
    if (token == “)”) token = nextToken();
      else syntaxError();
    return type;                 // <Exp> の型
          :
```

式の型判定

- <Unsigned> の型 (変数, 配列)

```
case 名前 :                    // 変数の場合
    String name = token.strValue の値; // 変数名を得る
    token = nextToken();
    Type type = varTable.getType (name);  // 変数表に登録されている型
    if (token == "[" ) {                  // 配列の場合
      "[" <Exp> "]" の処理;
      switch (type) {
        case ARRAYOFINT :        type = INT;       break
        case ARRAYOFDOUBLE : type = DOUBLE; break;
        case ARRAYOFCHAR :     type = CHAR;     break;
                  :
        default : syntaxError();
      }
    }
    return type;
```

スカラ型に変換

演算によって得られる式の型

- 式の型判定

```
/** @return type1op type2 の演算によって得られる型 */
Type expType (Op op, Type type1, Type type2)
```

例 : double 型 - int 型 の演算によって得られる型

```
expType (“-”, DOUBLE, INT)
```

演算子 op を適用できない被演算子型の場合は
エラー識別用の型 Type.ERR を返す

演算によって得られる式の型

- 各演算につき式の型の表を用意しておく

*	int	long	double	char	String
int	int	long	double	int	ERR
long	long	long	double	long	ERR
double	double	double	double	double	ERR
char	int	long	double	int	ERR
String	ERR	ERR	ERR	ERR	ERR

expType (“*”, INT, DOUBLE) = DOUBLE

式の型判定

- <Term> の型

```
Type parseTerm() {
   Type type1 = parseFactor();        // <Factor> の型を記憶
   while (token == "*") {
      token = nextToken();
      if (token ∈ First (<Factor>)) {
          Type type2 = parseFactor();
          type1 = expType ("*", type1, type2);
                                            // type1 * type2 の型を判別
          if (type1 == ERR)          // 演算子を適用できる型か
              syntaxError ("型の整合性が取れていません")
      } else SyntaxError();
   }
  return type1;
}
```

式解析部の返り値

- 式解析部の返り値
  - 左辺値の有無が必要な場合 : boolean
  - 式の型が必要な場合 : Type

どちらも必要な場合は？

```
class ExpType {
  boolean hasLeftValue; // 左辺値の有無
  Type type;            // 式の型
}
```

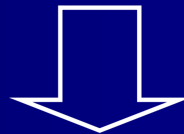
式解析部の返り値

```
ExpType parseUnsigned () {
    switch (token) {
        case 名前 :                    // 変数の場合
            String name = token.strValue の値; // 変数名を得る
            Type type = varTable.getType (name); // 変数表参照
            token = nextToken();
            return new ExpType (true, type);  // 左辺値あり, 変数の型
        case 整数 :                    // 整数の場合
            token = nextToken();
            return new ExpType (false, INT);  // 左辺値無し, int型
                :
}
```

break 文, case 値ラベル

- break 文 : ループ, switch 文からの脱出
- continue 文 : 次のループへ
- case 値, default ラベル : switch 文の分岐

ループ, switch文内部でのみ使用可能



ループ内部, switch文内部の判定が必要

ループ内部の判定

boolean型のフィールド変数を準備

```
private boolean inLoop = false;    /* ループ内部にいるか？ */
private boolean inSwitch = false;  /* swich文内部にいるか？ */
```

初期値は false

ループに入ったときに
inLoop の値を true にする

ループ内部の判定

```
private boolean inLoop = false;    /* ループ内部にいるか？ */
```

```
parseWhile() {
   if (token == “while”) token = nextToken(); else syntaxError();
   if (token == “(”) token = nextToken(); else syntaxError();
   if (token ∈ first (<Exp>)) parseExp(); else syntaxError();
   if (token == “)”) token = nextToken(); else syntaxError();
   if (token ∈ first (<St>)) {
      boolean outerLoop = inLoop; /* while文外部の情報を記憶 */
      inLoop = true;          /* フィールド変数の値をループ内部に */
      parseSt();       /* この<St>内はループ内部として処理される */
      inLoop = outerLoop;              /* 外部のループ情報を復帰 */
   } else syntaxError();
}
```

ループ内部の判定

```
parseSt() {
    if (token ∈ first(<If>))                parseIf();
    else if (token ∈ first (<While>)) parseWhile();
    else if (token ∈ first (<Exp>))    parseExp();
    ease if (token == “break”) {        /* break文か？ */
        if (inLoop || inSwitch)      /* ループ or switch 文内か？ */
            parseBreak();              /* break文の解析へ */
        else syntaxError (“ループ内ではありません”);
} else ...
                :
```

ループ, switch 文の外で break 文が来たらエラー

Warning 検査

- Warning :
  - 文法上はエラーではないが
    プログラマのミスの可能性が高い

```
int x, y, z;
while (true) {
    x = y;
    break;
    print (x);
}
x + 1;
if (x == 1);
```

- int x, y, z; → 変数 z は一度も使用されない
- x = y; → これ以前に y に値が代入されていない
- print (x); → この文は絶対に実行されない
- x + 1; → 代入も出力もされない式文
- if (x == 1); → else 節の無い if 文の文が空文

Warning 処理

Warning 時は警告メッセージを出してコンパイルを続ける

```
private void warning (String err_mes) {
    System.out.println (analyzeAt() + “で警告”);
      /* LexicalAnalyzer の analyzeAt() を用いて警告位置表示 */
    System.out.println (err_mes);  /* 警告メッセージ表示 */
    /* そのままコンパイルを継続 */
}
```

変数の未代入, 未参照

- 未代入
  - 値の代入されていない変数の値を参照
- 未参照
  - プログラム中1度も値が参照されない

```
public class Var {
    private Type type;              // 型
    private String name;            // 変数名
    private int address;            // 番地
    private int size;               // 配列のサイズ
    private boolean assigned;  // 代入されたか？
    private boolean reffered;  // 参照されたか？
```

```
boolean parseUnsigned () {
  switch (token) {
    case NAME :                     // 変数の場合
      String name = token.strValue の値;    // 変数名を得る
      Var var = varTable.getVar (name); // 変数を得る
      var.reffered = true;                      // 参照された
      token = nextToken();
      if (token == “=”) {          // 次に来るのが代入の場合
        var.assigned = true;                  // 代入された
      } else {       // 代入ではない = 右辺値が求められている
        if (!var.assigned)
          warning (name + “は値が代入されていません”);
      }
                 :
}
```

```
void parseProgram () {
   if (token ∈ First (<MainFunction>))
      parseMainFunction();
      else syntaxError ();
   if (token == “$”) {
      コンパイル完了処理
      for (Var var : varTable.varList) { // 各変数に対して処理
        if (!var.reffered) // 最後まで参照されていない
          warning (var.name + “は一度も参照されていません”);
      }
   } else syntaxError (“ファイル末ではありません”);
}
```