Compilers (CS30003)

Lecture 21-22

Pralay Mitra

Autumn 2023-24

Declaration Grammar

0: $P \rightarrow D$
1: $D \rightarrow T\ V\ ;\ D$
2: $D \rightarrow \epsilon$
3: $V \rightarrow V\ ,\ \textbf{id}$
4: $V \rightarrow \textbf{id}$
5: $T \rightarrow B$
6: $B \rightarrow \textbf{int}$
7: $B \rightarrow \textbf{float}$

int X, Y;

Name	Type	Size	Offset
X	int	4	0
Y	int	4	4

Attributes for Types

- *type*: Type expression for *B*, *T*.
  This is an inherited attribute.
- *width*: The width (storage units in bytes) of a type *B*, *T*.
  This is an inherited attribute.
- *t*: Global to pass the *type* information from a *B* node to the node for production *V* → **id**.
- *w*: Global to pass the *width* information from a *B* node to the node for production *V* → **id**.

Declaration Grammar with Semantic Actions

0:	P	→	D	{ $offset = 0$; }
1:	D	→	$T\ V$; D_1	
2:	D	→	ϵ	
3:	V	→	V , **id**	{ $update(\mathbf{id}.loc, t, w, offset)$; $offset = offset + w$; }
4:	V	→	**id**	{ $update(\mathbf{id}.loc, t, w, offset)$; $offset = offset + w$; }
5:	T	→	B	{ $t = B.type$; $w = B.width$; $T.type = B.type$; $T.width = B.width$; }
6:	B	→	**int**	{ $B.type = integer$; $B.width = 4$; }
7:	B	→	**float**	{ $B.type = float$; $B.width = 8$; }

Attributes

	Inherited Attributes	Synthesized Attributes
0	$P \rightarrow D$	$P \rightarrow D$
1	$D \rightarrow T\ V\ ;\ D$	$D \rightarrow V\ ;\ D$
2	$D \rightarrow \varepsilon$	$D \rightarrow \varepsilon$
3	$V \rightarrow V$, **id**	$V \rightarrow V$, **id**
4	$V \rightarrow$ **id**	$V \rightarrow T$ **id**
5	$T \rightarrow B$	$T \rightarrow B$
6	$B \rightarrow$ **int**	$B \rightarrow$ **int**
7	$B \rightarrow$ **float**	$B \rightarrow$ **float**

int X, Y;

Name	Type	Size	Offset
X	int	4	0
Y	int	4	4

Semantic Actions using Synthesized Attributes

0: $P \rightarrow \{\ offset = 0;\ \}\ D$

1: $D \rightarrow V\ ;\ D_1$

2: $D \rightarrow \epsilon$

3: $V \rightarrow V_1$, **id**
$\{\ update(\textbf{id}.loc, V_1.type, V_1.width, offset);$
$offset = offset + T.width;$
$V.type = V_1.type;\ V.width = V_1.width;\ \}$

4: $V \rightarrow T$ **id**
$\{\ update(\textbf{id}.loc, T.type, T.width, offset);$
$offset = offset + T.width;$
$V.type = T.type;\ V.width = T.width;\ \}$

5: $T \rightarrow B$
$\{\ T.type = B.type;\ T.width = B.width;\ \}$

6: $B \rightarrow$ **int** $\{\ B.type = integer;\ B.width = 4;\ \}$

7: $B \rightarrow$ **float** $\{\ B.type = float;\ B.width = 8;\ \}$

Use in Translation

- $E \rightarrow E_1 + E_2$
- $E \rightarrow$ id

```
int a, b, c;
a=b+c;

100: t1=b+c
101: a=t1
```

```
int a,b;
float c;
a=b+c

100:  t1=int2flt(b)
101:  t2=t1+c
102:  t3=float2int(t2)
103:  a=t3
```

Use in translation

$E \rightarrow E_1 + E_2$	{ *E.loc=gentemp*(); *if*(E_1.*type*!=E_2.*type*) *update*(*E.loc*, float, *offset*); *t=gentemp*(); *update*(*t*, float, *offset*); *if*(E_1.*type*==int) *emit*(*t*, '=', int2flt(E_1.*loc*)); *emit*(*E.loc*, '=', *t*, '+', E_2.*loc*); *else* *emit*(*t*, '=', int2flt(E_2.*loc*)); emit(*E.loc*, '=', E_1.*loc*, '+', *t*); *endif* *else* *update*(*E.loc*, E_1,*type*, *offset*); *emit*(*E.loc*, '=', E_1.*loc*, '+', E_2.*loc*); }
$E \rightarrow$ id	{ *E.loc* = id.*loc*;}

Use in translation

convInt2Bool(E):

```
if(E.type==int)
    E.falselist=makelist(nextinstr);
    emit(if, E.loc, '=', 0 goto …);
    E.truelist=makelist(nextinstr);
    emit(goto …);
end
```

Use in translation – Homework

Grammar:

$E \rightarrow E_1 < E_2$

$E \rightarrow E_1\ N_1\ ?\ M_1\ E_2\ N_2\ :\ M_2\ E_3$

$M \rightarrow \varepsilon$

$N \rightarrow \varepsilon$

Translate:

```
int a, b, c, d;
d = a – b !=  0? b + c : b – c;
d = a – b ? b + c : b – c;
```

Types and Declaration

- Type expression
  - int [2][3]
- Type Equivalence
- Declaration
  - $P \rightarrow D$
  - $D \rightarrow T$ **id** ; $D \mid \varepsilon$
  - $T \rightarrow B\ C \mid$ **struct** '{' D '}'
  - $B \rightarrow$ **int** | **float**
  - $C \rightarrow$ **[num]** $C \mid \varepsilon$

Attributes for Types

- *type*: Type expression for *B*, *C*.
  This is a synthesized attribute.
- *width*: The width (storage units in bytes) of a type *B*, *C*.
  This is a synthesized attribute.
- *t*: Variable to pass the *type* information from a *B* node to the node for production $C \rightarrow \varepsilon$.
  This is an inherited attribute.
- *w*: Variable to pass the *width* information from a *B* node to the node for production $C \rightarrow \varepsilon$.
  This is an inherited attribute.

Types and Declaration with semantic action

- Storage Layout for local names

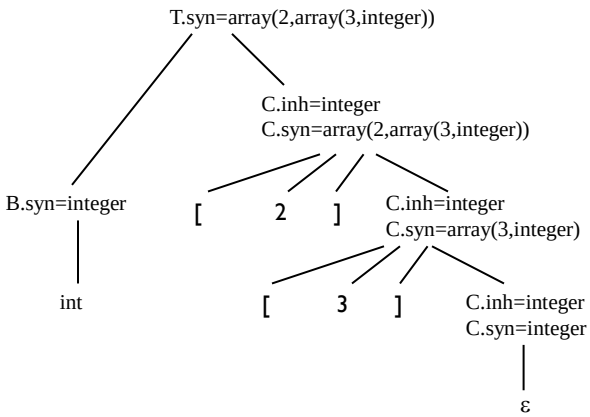
PRODUCTIONS	TYPES AND WIDTHS
$T \rightarrow B$	{t=B.type; w=B.width;}
C	{T.type=C.type; T.width=C.width;}
$B \rightarrow$ **int**	{ B.type=integer; B.width=4; }
$B \rightarrow$ **float**	{ B.type=float; B.width=8; }
$C \rightarrow \varepsilon$	{C.type=t; C.width=w;}
$C \rightarrow$ [**num**] C_1	{ C.type=array(num.val, C1.type); C.width=num.val $\times$ C1.width; }

int [2][3]

An example

Among *type* and *width* which one is *synthesized* and which one is *inherited*?

int [2][3]



Expression Grammar with Arrays

1. $S \rightarrow \text{id} = E\ ;$
2. $S \rightarrow A = E\ ;$
3. $E \rightarrow E_1 + E_2$
4. $E \rightarrow \text{id}$
5. $E \rightarrow A$
6. $A \rightarrow \text{id}\ [\ E\ ]$
7. $A \rightarrow A_1\ [\ E\ ]$

Input:

```
int a[2][3], b, c;
b = c + a[i][j];
```

Output:

```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a[t3]
t5 = c + t4
b  =  t5
```

Attributes for Arrays

- *A.loc*
  - Temporary used for computing the offset for the array reference by summing the terms $i_j * W_j$
- *A.array*
  - Pointer to the symbol-table entry for the array name with base and type. Base address of the array (*A.array.base*) is used to determine the actual *l-value* of an array reference after all the index expressions are analysed.
- *A.type*
  - Type of the sub-array generated by A. For any type *t*, the width is given by *t.width*. We use types as attributes, rather than widths, since types are needed anyway for type checking. For any array type *t*, suppose that *t.elem* gives the element type.

Expression Grammar with Semantic Actions

$S \rightarrow$ id = E ; { emit(id.loc,'=',E.loc); }

$S \rightarrow A = E$; { emit(A.array.base,'[',A.loc,']','=',E.loc); }

$E \rightarrow E_1 + E_2$ { *E.loc=gentemp();*
emit(E.loc,'=',E_1.loc,'+',E_2.loc); }

$E \rightarrow$ id { *E.loc=id.loc;* }

$E \rightarrow A$ { *E.loc=gentemp();*
emit(E.loc,'=',A.array.base,'[',A.loc,']'); }

$A \rightarrow$ id [E] { A.array=lookup(id);
A.type=A.array.type.elem;
A.loc=gentemp();
emit(A.loc,'=',E.loc,'*',A.type.width); }

$A \rightarrow A_1$ [E] { A.array=A_1.array;
A.type=A_1.type.elem;
t=gentemp();
A.loc=gentemp();
emit(t,'=',E.loc,'*',A.type.width);
emit(A.loc,'=',A_1.loc,'+',t); }

Expression Grammar with Arrays

Input:

```
int a[2][3], b, c[5];
int i,j,k;

b = c[k] + a[i][j];
```

Output:

```
t1 = k * 4
t2 = c[t1]
t3 = i * 12
t4 = j * 4
t5 = t3 + t4
t6 = a[t5]
t7 = t2 + t6
b  = t7
```

Symbol Table

Name	Type	Size	Offset
a	array(2, array(3, int))	24	0
b	int	4	24
c	array(5, int)	20	28
i	int	4	48
j	int	4	52
k	int	4	56

Declaration Grammar for Type Expression

Inherited Attribute	Synthesized Attribute
0: $P \rightarrow D$	0: $P \rightarrow D$
1: $D \rightarrow T\ V\ ;\ D$	1: $D \rightarrow V\ ;\ D$
2: $D \rightarrow \epsilon$	2: $D \rightarrow \epsilon$
3: $V \rightarrow V\ ,\ \mathbf{id}\ C$	3: $V \rightarrow V\ ,\ \mathbf{id}\ C$
4: $V \rightarrow \mathbf{id}\ C$	4: $V \rightarrow T\ \mathbf{id}\ C$
5: $T \rightarrow B$	5: $T \rightarrow B$
6: $B \rightarrow \mathbf{int}$	6: $B \rightarrow \mathbf{int}$
7: $B \rightarrow \mathbf{float}$	7: $B \rightarrow \mathbf{float}$
8: $C \rightarrow [\ \mathbf{num}\ ]\ C$	8: $C \rightarrow [\ \mathbf{num}\ ]\ C$
9: $C \rightarrow \epsilon$	9: $C \rightarrow \epsilon$

Example:

```
int a, b;
int x, y[10], z;
float w[5];
```

Symbol Table

Name	Type	Size	Offset
a	int	4	0
b	int	4	4
x	int	4	8
y	array(10, int)	40	12
z	int	4	52
w	array(5, float)	8	56

Types and Declaration

- Sequences of declarations

$P \rightarrow$ D	{ offset=0; }
$D \rightarrow T$ **id**; D_1	{ *update*(id.lexval, T.type, offset); offset = offset+T.width;}
$D \rightarrow \varepsilon$	

* Fields in structures
  * float nodeval;
  * struct node { float nodeval; char nodetype; int child};
  * struct leaf { float nodeval; char nodetype; };

Functions

Function Definition Grammar

1:	D	$\rightarrow$	T id (F_{opt});	{ *insert*(ST_{gbl}, id, T.*type*, function, F_{opt}.ST); }
2:	F_{opt}	$\rightarrow$	F	{ F_{opt}.ST = F.ST; }
3:	F_{opt}	$\rightarrow$	ϵ	{ F_{opt}.ST = 0; }
4:	F	$\rightarrow$	F_1 , T id	{ F.ST = F_1.ST; *insert*(F.ST, id, T.*type*, 0); }
5:	F	$\rightarrow$	T id	{ F.ST = *CreateSymbolTable*(); *insert*(F.ST, id, T.*type*, 0); }
6:	T	$\rightarrow$	int	{ T.*type* = int }
7:	T	$\rightarrow$	double	{ T.*type* = double }
8:	T	$\rightarrow$	void	{ T.*type* = void }

```
int func(int i, double d);
```

ST(global) *This is the Symbol Table for global symbols*

Name	Type	Init. Val.	Size	Offset	Nested Table
func	*function*	null	0	...	ptr-to-ST(func)

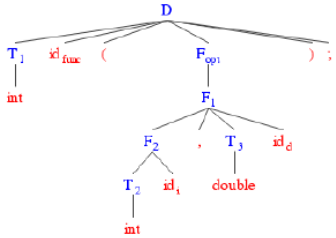
ST(func) *This is the Symbol Table for function* func

Name	Type	Init. Val.	Size	Offset	Nested Table
i	int	null	4	0	null
d	double	null	8	4	null
retVal	int	null	4	12	null

Function Declaration Example

```
int func(int i, double d);
```

```
T1.type = int
T2.type = int
F2.ST = ST(func)
T3.type = dbl
F1.ST = ST(func)
F_opt.ST = ST(func)
```



ST(global)

Name	Type	Size	Offset	Nested Table
func	int × dbl → int	0	...	ST(func)

ST(func)

Name	Type	Size	Offset	Nested Table
i	int	4	0	null
d	dbl	8	4	null
_rv	int	4	12	null

Function Invocation Grammar

	0:	D	$\rightarrow$	T **id** (F_{opt}) { L }
1 |	2:	L	$\rightarrow$	L_1 S | S
	3:	S	$\rightarrow$	**return** E ; { Check if function.*type* matches E.*type*; *emit*(**return** E.*loc*); }
	4:	E	$\rightarrow$	**id** (A_{opt}) { ST = *lookup*(ST_{gbl}, **id**).*symtab*; For every param p in A_{opt}.*list*; Match p.*type* with param type in ST; *emit*(**param** p.*loc*); E.*loc* = *gentemp*(*lookup*(ST_{gbl}, **id**).*type*); *emit*(E.*loc* = **call id**, *length*(A_{opt}.*list*)); }
	5:	A_{opt}	$\rightarrow$	A { A_{opt}.*list* = A.*list*; }
	6:	A_{opt}	$\rightarrow$	ϵ { A_{opt}.*list* = 0; }
	7:	A	$\rightarrow$	A_1 , E { A.*list* = *Merge*(A_1.*list*, *Makelist*(E.*loc*, E.*type*)); }
	8:	A	$\rightarrow$	E { A.*list* = *Makelist*(E.*loc*, E.*type*); }

```
int a, b, c;
double d, e;
...
a = func(b + c, d * e);
return a;
```

List of Params

t1	int
t2	double

```
t1 = b + c
t2 = d * e
param t1
param t2
t3 = call func, 2
a = t3
```

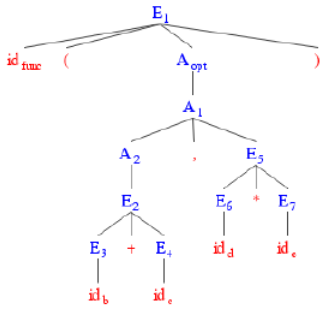
Function Invocation Example

```
int a, b, c;
double d, e;
...
a = func(b + c, d * e);
return a;

-------------------------

t1 = b + c
t2 = d * e
param t1
param t2
t3 = call func, 2
```

```
E3.loc = b, E3.type = int
E4.loc = c, E4.type = int
E2.loc = t1, E2.type = int
A2.list = {t1}
E6.loc = d, E6.type = dbl
E7.loc = e, E7.type = dbl
E5.loc = t2, E5.type = dbl
A1.list = {t1, t2}
A_opt.list = {t1, t2}
E1.loc = t3, E1.type = int
```



ST(global)

Name	Type	Size	Offset	Nested Table
func	int × dbl → int	0	...	ST(func)

ST(func)

Name	Type	Size	Offset	Nested Table
i	int	4	0	null
d	dbl	8	4	null
__rv	int	4	12	null

ST(?)

Name	Type	Size	Offset	Nested Table
a	int	4	0	null
b	int	4	4	null
c	int	4	8	null
d	dbl	8	16	null
e	dbl	8	24	null
t1	int	4	28	null
t2	dbl	8	32	null
t3	int	4	40	null

Lexical Scope Management

Grammar for Global, Function and Nested Block Scopes

0:	Pgm	$\rightarrow$	TU	{ $UpdateOffset(ST_{gbl})$; } // End of TAC Translate
1:	TU	$\rightarrow$	$TU_1\ P$	
2:	TU	$\rightarrow$	$M\ P$	
3:	M	$\rightarrow$	ϵ	{ $ST_{gbl} = CreateSymbolTable()$; $ST_{gbl}.parent = 0$; $cST = ST_{gbl}$; }
4:	P	$\rightarrow$	VD	// Variable Declaration
5:	P	$\rightarrow$	PD	// Function Prototype Declaration
6:	P	$\rightarrow$	FD	// Function Definition
7:	VD	$\rightarrow$	$T\ V$;	{ $type_{gbl} = null$; $width_{gbl} = 0$; }
8:	V	$\rightarrow$	V_1 , **id** C	{ $Name = lookup(cST,$ **id**$)$; $Name.category = (cST == ST_{gbl})$? global: local; $Name.type = C.type$; $Name.size = C.width$; }
9:	V	$\rightarrow$	**id** C	{ $Name = lookup(cST,$ **id**$)$; $Name.category = (cST == ST_{gbl})$? global: local; $Name.type = C.type$; $Name.size = C.width$; }
10:	C	$\rightarrow$	**[num]** C_1	{ $C.type = array($**num**$.value, C_1.type)$; $C.width =$**num**$.value \times C_1.width)$; }
11:	C	$\rightarrow$	ϵ	{ $C.type = type_{gbl}$; $C.width = width_{gbl}$; }
12:	T	$\rightarrow$	B	{ $type_{gbl} = T.type = B.type$; $width_{gbl} = T.width = B.width$; }
13:	B	$\rightarrow$	**int**	{ $B.type =$ int; $B.width =$ sizeof$(B.type)$; }
14:	B	$\rightarrow$	**double**	{ $B.type =$ double; $B.width =$ sizeof$(B.type)$; }
15:	B	$\rightarrow$	**void**	{ $B.type =$ void; $B.width =$ sizeof$(B.type)$; }

Grammar for Global, Function and Nested Block Scopes

16:	PD	$\rightarrow$	$T\ FN$ (FP_{opt});	{ $UpdateOffset(cST)$; $cST = cST.parent$; }
17:	FD	$\rightarrow$	$T\ FN$ (FP_{opt}) CS	{ $UpdateOffset(cST)$; $cST = cST.parent$; }
18:	FN	$\rightarrow$	**id**	{ $Name = lookup(ST_{gbl},$ **id**$)$; $ST = Name.symtab$; if (ST is $null$) $ST = CreateSymbolTable()$; $ST.parent = ST_{gbl}$; $Name.category =$ function; $Name.symtab = ST$; endif $cST = ST$; }
19:	FP_{opt}	$\rightarrow$	FP	
20:	FP_{opt}	$\rightarrow$	ϵ	
21:	FP	$\rightarrow$	FP_1 , T **id**	{ $Name = lookup(cST,$ **id**$)$; $Name.category =$ param; $Name.type = T.type$; $Name.size = T.width$; }
22:	FP	$\rightarrow$	T **id**	{ $Name = lookup(cST,$ **id**$)$; $Name.category =$ param; $Name.type = T.type$; $Name.size = T.width$; }
23:	CS	$\rightarrow$	{ $N\ L$ }	{ $UpdateOffset(cST)$; $cST = cST.parent$; }
24:	N	$\rightarrow$	ϵ	{ if ($cST.parent$ is not ST_{gbl}) // Not a function scope $N.ST = CreateSymbolTable()$; $N.ST.parent = cST$; $cST = N.ST$; endif }
25:	L	$\rightarrow$	$L_1\ S$	// List of Statements – Statement actions not shown
26:	L	$\rightarrow$	LD	
27:	LD	$\rightarrow$	$LD_1\ VD$	// List of Declarations
28:	LD	$\rightarrow$	ϵ	

Grammar for Global, Function and Nested Block Scopes

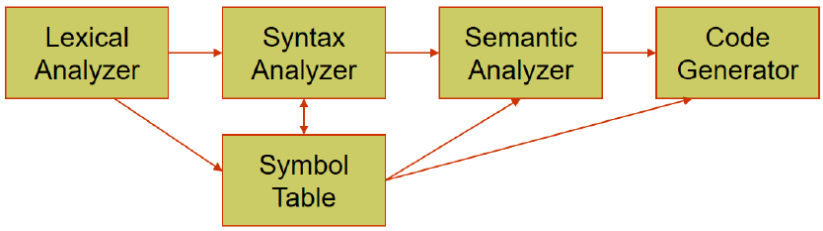
29:	S	$\rightarrow$	CS	
30:	S	$\rightarrow$	E ;	
31:	S	$\rightarrow$	**return** E ;	{ *emit*(**return** E.*loc*); }
32:	S	$\rightarrow$	**return** ;	{ *emit*(**return**); }
33:	E	$\rightarrow$	$E_1 = E_2$	{ E.*loc* = *gentemp*(); *emit*(E_1.*loc* '=' E_2.*loc*); *emit*(E.*loc* '=' E_1.*loc*); }
34:	E	$\rightarrow$	**id**	{ E.*loc* =**id**.*loc*; }
35:	E	$\rightarrow$	**num**	{ E.*loc* = *gentemp*(); *emit*(E.*loc* = **num**.*val*); }
36:	E	$\rightarrow$	AR	{ E.*loc* = *gentemp*(); *emit*(E.*loc* '=' AR.*array*.*base* '[' AR.*loc* ']'); }
37:	AR	$\rightarrow$	**id** [E]	{ AR.*array* = *lookup*(*cST*, **id**); AR.*type* = AR.*array*.*type*.*elem*; AR.*loc* = *gentemp*(); *emit*(AR.*loc* '=' E.*loc* '*' AR.*type*.*width*); }
38:	AR	$\rightarrow$	AR_1 [E]	{ AR.*array* = AR_1.*array*; AR.*type* = AR_1.*type*.*elem*; t = *gentemp*(); AR.*loc* = *gentemp*(); *emit*(t '=' E.*loc* '*' AR.*type*.*width*); *emit*(AR.*loc* '=' AR_1.*loc* '+' t); }

Grammar for Global, Function and Nested Block Scopes

39:	E	$\rightarrow$	**id** (AP_{opt})	{ ST = *lookup*(ST_{gbl}, **id**).*symtab*; For every param p in AP_{opt}.*list*; Match p.*type* with param type in ST; *emit*(**param** p.*loc*); E.*loc* = *gentemp*(*lookup*(ST_{gbl}, **id**).*type*); *emit*(E.*loc* = **call id**, *length*(AP_{opt}.*list*)); }
40:	AP_{opt}	$\rightarrow$	AP	{ AP_{opt}.*list* = AP.*list*; }
41:	AP_{opt}	$\rightarrow$	ϵ	{ AP_{opt}.*list* = 0; }
42:	AP	$\rightarrow$	AP_1 , E	{ AP.*list* = *Merge*(AP_1.*list*, *Makelist*((E.*loc*, E.*type*)); }
43:	AP	$\rightarrow$	E	{ AP.*list* = *Makelist*((E.*loc*, E.*type*)); }

Symbol Table

Symbol Table



Symbol Table: Entries

An ST stores varied information about identifiers:
- Name (as a string)
  - ▷ Name may be qualified for scope or overload resolution
- Data type (explicit or pointer to Type Table)
- Block level
- Scope (global, local, parameter, or temporary)
- Offset from the base pointer (for local variables and parameters only) – to be used in Stack Frames
- Initial value (for global and local variables), default value (for parameters)
- Others (depending on the context)

A Name (Symbol) may be any one of:
- Variable (user-define / unnamed temporary)
- Constant (String and non-String)
- Function / Method (Global / Class)
- Alias
- Type – Class / Structure / Union
- Namespace

Symbol Table: Scope Rules

- Scoping of Symbols may be static (compile time) or dynamic (run time)

Static Scoping

```
const int b = 5;

int foo() { // Uses lexical context for b
    int a = b + 5; // b in global
    return a;
}

int bar() {
    int b = 2;
    return foo();
}

int main() {
    foo(); // returns 10
    bar(); // returns 10

    return 0;
}
```

- Used in C / C++ / Java – run-time polymorphism in C++ is an exception
- Good for compilers
- Needs symbol table at compile-time only

Dynamic Scoping

```
const int b = 5;

int foo() { // Uses run-time context for b
    int a = b + 5; // b in bar
    return a;
}

int bar() {
    int b = 2;
    return foo();
}

int main() {
    foo(); // returns 10
    bar(); // returns 7

    return 0;
}
```

- Used in Python / Lisp
- Good for interpreters
- Needs symbol table at compile-time as well as run-time

Symbol Table: Scope and Visibility

Scope (visibility) of identifier = portion of program where identifier can be referred to

Lexical scope = textual region in the program

- Statement block
- Method body
- Class body
- Module / package / file
- Whole program (multiple modules)

Symbol Table: Scope and Visibility

- Global scope
  - Names of all classes defined in the program
  - Names of all global functions defined in the program
- Class scope
  - Instance scope: all fields and methods of the class
  - Static scope: all static methods
  - Scope of subclass nested in scope of its superclass
- Method scope
  - Formal parameters and local variables in code block of body method
- Code block scope
  - Variables defined in block

Symbol Table: Interface

- Create Symbol Table
- Search (lookup)
- Insert
- Search & Insert
- Update Attribute

Symbol Table: Implementation

- Linear List
- Hash Table
- Binary Search Tree

Example: Global & Function Scopes

```
int m_dist(int x1, int y1, int x2, int y2) {
    int d, x_diff, y_diff;
    x_diff = (x1 > x2) ? x1 - x2 : x2 - x1;
    y_diff = (y1 > y2) ? y1 - y2 : y2 - y1;
    d = x_diff + y_diff;
    return d;
}
int x1 = 0, y1 = 0; // Global static
int main(int argc, char *argv[]) {
    int x2 = -2, y2 = 3, dist = 0;
    dist = m_dist(x1, y1, x2, y2);
    return 0;
}
```

```
m_dist:
   if x1 > x2 goto L1
   t1 = x2 - x1
   goto L2
L1:t1 = x1 - x2
L2:x_diff = t1
   if y1 > y2 goto L3
   t2 = y1 - y2
   goto L4
L3:t2 = y2 - y1
L4:y_diff = t2
   d = x_diff + y_diff
   return d
```

```
// global initialization
   x1_g = 0
   y1_g = 0
main:
   x2 = -2
   y2 = 3
   dist = 0
   param y2
   param x2
   param y1_g
   param x1_g
   dist = call m_dist, 4
   return 0
```

ST.glb				Parent: *Null*
m_dist	int × int × int × int → int			
		func	0	0
x1_g	int	global	4	
y1_g	int	global	4	
main	int × arr(*,char*) → int			
		func	0	0

ST.m_dist()				Parent: *ST.glb*
y2	int	param	4	+20
x2	int	param	4	+16
y1	int	param	4	+12
x1	int	param	4	+8
d	int	local	4	−4
x_diff	int	local	4	−8
y_diff	int	local	4	−12
t1	int	temp	4	−16
t2	int	temp	4	−20

ST.main()				Parent: *ST.glb*
argv	arr(*,char*)			
		param	4	+8
argc	int	param	4	+4
x2	int	local	4	−4
y2	int	local	4	−8
dist	int	local	4	−12

- *Cols: Name, Type, Category, Size, Offset*
- *For a function* `T f(T1, T2)` *the type is:* $T1 \times T2 \rightarrow T$
- *Base pointer is 0*
- *Return address and Return value are not shown*
- *Symbol Tables form a tree with ST.glb as the root*

Example: Global, Function & Block Scopes

```
int m_dist(int x1, int y1, int x2, int y2) {
    int d, { int x_diff, \\ Nested block
    { int y_diff; \\ Nested nested block
    x_diff = (x1 > x2) ? x1 - x2 : x2 - x1;
    y_diff = (y1 > y2) ? y1 - y2 : y2 - y1;
    } }
    d = x_diff + y_diff;
    return d;
}
int x1 = 0, y1 = 0; // Global static
int main(int argc, char *argv[]) {
    int x2 = -2, y2 = 3, dist = 0;
    dist = m_dist(x1, y1, x2, y2);
    return 0; }
```

```
m_dist:
   if x1 > x2 goto L1
   t1 = x2 - x1
   goto L2
L1:t1 = x1 - x2
L2:x_diff_$2 = t1
   if y1 > y2 goto L3
   t2 = y1 - y2
   goto L4
L3:t2 = y2 - y1
L4:y_diff_$1 = t2
   d = x_diff + y_diff
   return d
```

```
// global initialization
   x1_g = 0
   y1_g = 0
main:
   x2 = -2
   y2 = 3
   dist = 0
   param y2
   param x2
   param y1_g
   param x1_g
   dist = call m_dist, 4
   return 0
```

ST.glb				Parent: *Null*
m_dist	int × int × int × int → int			
		func	0	0
x1_g	int	global	4	0
y1_g	int	global	4	−4
main	int × arr(*,char*) → int			
		func	0	0

ST.m_dist()				Parent: *ST.glb*
y2	int	param	4	+20
x2	int	param	4	+16
y1	int	param	4	+12
x1	int	param	4	+8
d	int	local	4	−4
x_diff_$2	int	local	4	−8
y_diff_$1	int	local	4	−12
t1	int	temp	4	−16
t2	int	temp	4	−20

ST.m_dist().$2				Parent: *ST.m_dist()*
x_diff	int	local	4	0

ST.m_dist().$1				Parent: *ST.m_dist().$2*
y_diff	int	local	4	0

ST.main()				Parent: *ST.glb*
argv	arr(*,char*)			
		param	4	+8
argc	int	param	4	+4
x2	int	local	4	−4
y2	int	local	4	−8
dist	int	local	4	−12

Cols: Name, Type, Category, Size, Offset

- Static Allocation
- Automatic Allocation
- Embedded Automatic Allocation

Example: Global & Function Scopes, typedef

```
typedef struct { int _x, _y; } Point;
int m_dist(Point p, Point q) {
    int d, x_diff, y_diff;
    x_diff=(p._x>q._x)?p._x-q._x: q._x-p._x;
    y_diff=(p._y>q._y)?p._y-q._y: q._y-p._y;
    d = x_diff + y_diff;
    return d;
}
Point p = { 0, 0 };
int main() {
    Point q = { -2, 3 };
    int dist = 0;
    dist = m_dist(p, q);
    return 0;
}
```

```
m_dist:
   if p._x > q._x goto L1
   t1 = q._x - p._x
   goto L2
L1:t1 = p._x - q._x
L2:x_diff = t1
   if p._y > q._y goto L3
   t2 = q._y - p._y
   goto L4
L3:t2 = p._y - q._y
L4:y_diff = t2
   d = x_diff + y_diff
   return d
```

```
// global initialization
   x1_g = 0
   y1_g = 0
main:
   q._x = -2 // Offset(q)
   q._y = 3  // Offset(q+4)
   dist = 0
   param q
   param p
   dist = call m_dist, 2
   return 0
```

ST.glb				Parent: *Null*
m_dist	struct Point × struct Point → int			
		func	0	0
p_g	struct Point	global	8	
main	int × arr(*,char*) → int			
		func	0	0

ST.m_dist()				Parent: *ST.glb*
q	struct Point	param	8	+16
p	struct Point	param	8	+8
d	int	local	4	−4
x_diff	int	local	4	−8
y_diff	int	local	4	−12
t1	int	temp	4	−16
t2	int	temp	4	−20

ST_type.struct Point				Parent: *ST.glb*
_x	int	member	4	0
_y	int	member	4	−4

ST.main()				Parent: *ST.glb*
argv	arr(*,char*)			
		param	4	+8
argc	int	param	4	+4
q	struct Point	local	8	−12
dist	int	local	4	−20

Cols: Name, Type, Category, Size, Offset

Example: Global, Function & Class Scopes

```
class Point { public: int _x, _y;
    Point(int x, int y) : _x(x), _y(y) { }
    ~Point() {};
};
int m_dist(Point p, Point q) {
    int d, x_diff, y_diff;
    x_diff=(p._x>q._x)?p._x-q._x:q._x-p._x;
    y_diff=(p._y>q._y)?p._y-q._y:q._y-p._y;
    d = x_diff + y_diff;
    return d;
}
Point p = { 0, 0 };
int main(int argc, char *argv[]) {
    Point q = { -2, 3 };
    int dist = m_dist(p, q);
    return 0;
}
```

```
m_dist:
   if p._x > q._x goto L1
   t1 = q._x - p._x
   goto L2
L1:t1 = p._x - q._x
L2:x_diff = t1
   if p._y > q._y goto L3
   t2 = q._y - p._y
   goto L4
L3:t2 = p._y - q._y
L4:y_diff = t2
   d = x_diff + y_diff
   return d
```

C-tor / D-tor during Call / Return are not shown

```
crt: param 0 // Sys Caller
     param 0
     &p_g = call Point, 2
     param argv
     param argc
     result = call main, 2
     param &p_g
     call ~Point, 1
     return
main:param 3
     param -2
     &q = call Point, 2
     param q
     param p_g
     dist = call m_dist, 2
     param &q
     call ~Point, 1
     return 0
```

ST.glb				Parent: *Null*
m_dist	class Point × class Point → int			
		func	0	0
p_g	class Point	global	8	
main	int × arr(*,char*) → int			
		func	0	0

ST.m_dist()				Parent: *ST.glb*
q	class Point	param	8	+16
p	class Point	param	8	+8
d	int	local	4	−4
x_diff	int	local	4	−8
y_diff	int	local	4	−12
t1	int	temp	4	−16
t2	int	temp	4	−20

ST_type.class Point				Parent: *ST.glb*
_x	int	member	4	0
_y	int	member	4	−4
Point	int × int → class Point			
		method	0	0
~Point	class Point* → void			
		method	0	0

ST.main()				Parent: *ST.glb*
argv	arr(*,char*)	param	4	+8
argc	int	param	4	+4
q	class Point	local	8	−24
dist	int	local	4	−32

Cols: Name, Type, Category, Size, Offset

More Uses of Symbols Tables

- **String Table**: Various string constants
- **Constant Table**: Various non-string consts, const objects
- **Label Table**: Target labels
- **Keywords Table**: Initialized with keywords (KW)
  - KWs tokenized as id's and later marked as KWs on parsing
    - ▷ Simplifies lexical analysis
    - ▷ Good for languages where keywords are not reserved. *Note*: Keywords in C/C++ are reserved, while those in FORTRAN are not (how to know if an 'IF' is a keyword or an identifier?)
    - ▷ Good for languages like EDIF with user-defined keywords
- **Type Table**:
  - *Built-in Types*: `int`, `float`, `double`, `char`, `void` etc.
  - *Derived Types*: Types built with type builders like array, struct, pointer, enum etc. May need equivalence of type expressions like `int[]` & `int*`, separate tables etc.
  - *User-defined Types*: `class`, `struct` and `union` as types
  - *Type Alias*: `typedef`
  - *Named Scopes*: `namespace`

Example: Type Symbol Table

```
class Point { public: int _x, _y;
    Point(int x, int y) : _x(x), _y(y) {}
    ~Point() {};
};
class Rect { Point _lt, _rb; public:
    Rect(Point& lt, Point& rb):
        _lt(lt), _rb(rb) {}
    ~Rect() {}
    Point get_LT() { return _lt; }
    Point get_RB() { return _rb; }
};
```

```
int m_dist(Point p, Point q) {
    int d, x_diff, y_diff;
    x_diff=(p._x>q._x)?p._x-q._x:q._x-p._x;
    y_diff=(p._y>q._y)?p._y-q._y:q._y-p._y;
    d = x_diff + y_diff;
    return d;
}
Point p = { 0, 0 };
int main(int argc, char *argv[]) {
    Point q = { -2, 3 }; Rect r(p, q);
    int dist = m_dist(r.get_LT(), r.get_RB());
    return 0; }
```

ST.glb				Parent: *Null*
m_dist	class Point × class Point → int			
		func	0	0
p-g	class Point	global	8	
main	int × T_2d_Arr → int			
		func	0	0

ST.m_dist()			Parent:	*ST.glb*
q	class Point	param	8	+16
p	class Point	param	8	+8
d	int	local	4	−4
x_diff	int	local	4	−8
y_diff	int	local	4	−12
t1	int	temp	4	−16
t2	int	temp	4	−20

ST.main()			Parent:	*ST.glb*
argv	T_2d_Arr	param	4	+8
argc	int	param	4	+4
q	class Point	local	8	−24
dist	int	local	4	−32

ST_type.glb			Parent:	*Null*
Point	class Point		8	
Rect	class Rect		16	
T_2d_Arr	arr(*,char*)		4	

ST_type.class Point			Parent:	*ST_type.glb*
_x	int	member	4	0
_y	int	member	4	−4
Point	int × int → class Point			
~Point	class Point* → void			

ST_type.class Rect			Parent:	*ST_type.glb*
_lt	class Point	member	8	0
_rb	class Point	member	8	−8
Rect	class Point& × class Point& → class Rect	method	0	0
~Rect	class Rect* → void			
get_LT	class Rect* → class Point			
get_RB	class Rect* → class Point			

Cols: Name, Type, Category, Size, Offset

Example: main() & add(): Source & TAC

```
int add(int x, int y) {
    int z;
    z = x + y;
    return z;
}
void main(int argc,
          char* argv[]) {
    int a, b, c;
    a = 2;
    b = 3;
    c = add(a, b);
    return;
}
```

```
add:    t1 = x + y
        z = t1
        return z
main:   t1 = 2
        a = t1
        t2 = 3
        b = t2
        param a
        param b
        c = call add, 2
        return
```

ST.glb				
add	int × int → int	func	0	0
main	int × array(*, char*) → void			
		func	0	0

ST.add()				
y	int	param	4	+8
x	int	param	4	+4
z	int	local	4	0
t1	int	temp	4	−4

ST.main()				
argv	array(*, char*)			
		param	4	+8
argc	int	param	4	+4
a	int	local	4	0
b	int	local	4	−4
c	int	local	4	−8
t1	int	temp	4	−12
t2	int	temp	4	−16

Columns: Name, Type, Category, Size, & Offset

main() & add(): Peep-hole Optimized

```
int add(int x, int y) {
    int z;
    z = x + y;
    return z;
}
void main(int argc,
          char* argv[]) {
    int a, b, c;
    a = 2;
    b = 3;
    c = add(a, b);
    return;
}
```

```
add:    z = x + y
        return z
main:   a = 2
        b = 3
        param a
        param b
        c = call add, 2
        return
```

ST.glb

add	int × int → int	func	0	0
main	int × array(*, char*) → void	func	0	0

ST.add()

y	int	param	4	+8
x	int	param	4	+4
z	int	local	4	0

ST.main()

argv	array(*, char*)	param	4	+8
argc	int	param	4	+4
a	int	local	4	0
b	int	local	4	−4
c	int	local	4	−8

Columns: Name, Type, Category, Size, & Offset

Example: main() & d_add(): double type

```
double d_add(double x, double y) {
    double z;
    z = x + y;
    return z;
}
void main() {
    double a, b, c;
    a = 2.5;
    b = 3.4;
    c = d_add(a, b);
    return;
}
```

```
d_add:  z = x + y
        return z
main:   a = 2.5
        b = 3.4
        param a
        param b
        c = call d_add, 2
        return
```

ST.glb

d_add	dbl × dbl → dbl	function	0	0
main	void → void	function	0	0

ST.d_add()

x	dbl	param	8	0
y	dbl	param	8	16
z	dbl	local	8	24

ST.main()

a	dbl	local	8	0
b	dbl	local	8	8
c	dbl	local	8	16

Columns are: Name, Type, Category, Size, & Offset

Example: main() & swap()

```
void swap(int *x, int *y) {
    int t;
    t = *x;
    *x = *y;
    *y = t;
    return;
}
void main() {
    int a = 1, b = 2;
    swap(&a, &b);
    return;
}
```

```
swap:   t = *x;
        *x = *y;
        *y = t;
        return
main:   a = 1
        b = 2
        t1 = &a
        t2 = &b
        param t1
        param t2
        call swap, 2
        return
```

ST.glb				
swap	int* × int* → void	func	0	0
main	void → void	func	0	0
ST.swap()				
y	int*	prm	4	0
x	int*	prm	4	4
t	int	lcl	4	8

ST.main()				
a	int	lcl	4	0
b	int	lcl	4	4
t1	int*	lcl	4	8
t2	int*	lcl	4	12

Columns are: Name, Type, Category, Size, & Offset

Example: main() & C_add(): struct type

```
typedef struct {
    double re;
    double im;
} Complex;

Complex C_add(Complex x, Complex y) {
    Complex z;

    z.re = x.re + y.re;
    z.im = x.im + y.im;
    return z;
}

void main() {
    Complex a = { 2.3, 6.4 }, b = { 3.5, 1.4 }, c = { 0.0, 0.0 };
    c = C_add(a, b);
    return;
}
```

```
C_add:  z.re = x.re + y.re
        z.im = x.im + y.im
        *RV = z
        return
main:   a.re = 2.3
        a.im = 6.4
        b.re = 3.5
        b.im = 1.4
        c.re = 0.0
        c.im = 0.0
        param a
        param b
        c = call C_add, 2
        return
```

ST.glb: ST.glb.parent = null				
Complex	struct{dbl, dbl}	type	0	ST.Complex
C_add	Complex × Complex → Complex	function	0	ST.C_add
main	void → void	function	0	ST.main
ST.C_add(): ST.C_add.parent = ST.glb				
RV	Complex*	param	4	0
x	Complex	param	16	20
y	Complex	param	16	36
z	Complex	local	16	52

ST.Complex: ST.Complex.parent = ST.glb				
re	dbl	local	8	0
im	dbl	local	8	8
ST.main(): ST.main.parent = ST.glb				
a	Complex	local	16	0
b	Complex	local	16	16
c	Complex	local	16	32
RV	Complex	local	16	48

Columns are: Name, Type, Category, Size, & Offset

Example: main() & Sum(): Using Array & Nested Block

```
#include <stdio.h>

int Sum(int a[], int n) {
    int i, s = 0;
    for(i = 0; i < n; ++i) {
        int t;
        t = a[i];
        s += t;
    }
    return s;
}
void main() {
    int a[3];
    int i, s, n = 3;
    for(i = 0; i < n; ++i)
        a[i] = i;
    s = Sum(a, n);
    printf("%d\n", s);
}
```

```
Sum:    s = 0
        i = 0
L0:     if i < n goto L2
        goto L3
L1:     i = i + 1
        goto L0
L2:     t1 = i * 4
        t_1 = a[t1]
        s = s + t_1
        goto L1
L3:     return s
```

Block local variable `t` is named as `t_1` to qualify for the unnamed block within which it occurs.

```
main:   n = 3
        i = 0
L0:     if i < n goto L2
        goto L3
L1:     i = i + 1
        goto L0
L2:     t1 = i * 4
        a[t1] = i
        goto L1
L3:     param a
        param n
        s = call Sum, 2
        param "%d\n"
        param s
        call printf, 2
        return
```

Parameter `s` of `printf` is handled through varargs.

ST.glb: ST.glb.parent = null

Sum	array(*, int) × int → int	function	0	ST.Sum
main	void → void	function	0	ST.main

ST.main(): ST.main.parent = ST.glb

a	array(3, int)	local	12	0
i	int	local	4	12
s	int	local	4	16
n	int	local	4	20
t1	int	temp	4	24

ST.Sum(): ST.Sum.parent = ST.glb

a	int[]	param	4	0
n	int	param	4	4
i	int	local	4	8
s	int	local	4	12
t_1	int	local	4	16
t1	int	temp	4	20

Columns are: Name, Type, Category, Size, & Offset

Example: main(), function parameter & other functions

```
int trans(int a, int(*f)(int), int b)
{ return a + f(b); }

int inc(int x) { return x + 1; }

int dec(int x) { return x - 1; }

void main() {
    int x, y, z;

    x = 2;
    y = 3;
    z = trans(x, inc, y) +
        trans(x, dec, y);
    return;
}
```

```
trans:  param b
        t1 = call f, 1
        t2 = a + t1
        return t2

inc:    t1 = x + 1
        return t1

dec:    t1 = x - 1
        return t1
```

```
main:   x = 2
        y = 3
        param x
        param inc
        param y
        t1 = call trans, 3
        param x
        param dec
        param y
        t2 = call trans, 3
        z = t1 + t2
        return
```

ST.glb: ST.glb.parent = null

trans	int × ptr(int → int) × int → int	func	0	0
inc	int → int	func	0	0
dec	int → int	func	0	0
main	void → void	func	0	0

ST.trans(): ST.trans.parent = ST.glb

b	int	prm	4	0
f	ptr(int → int)	prm	4	4
a	int	prm	4	8
t1	int	tmp	4	12
t2	int	tmp	4	16

ST.inc(): ST.inc.parent = ST.glb

x	int	prm	4	0
t1	int	tmp	4	4

ST.dec(): ST.dec.parent = ST.glb

x	int	prm	4	0
t1	int	tmp	4	4

ST.main(): ST.main.parent = ST.glb

x	int	lcl	4	0
y	int	lcl	4	4
z	int	lcl	4	8
t1	int	tmp	4	12
t2	int	tmp	4	16

Columns are: Name, Type, Category, Size, & Offset

Example: Nested Blocks: Source & TAC

```
int a;
int f(int x) { // function scope f
    int t, u;
    t = x; // t in f, x in f
    { // un-named block scope f_1
        int p, q, t;
        p = a; // p in f_1, a in global
        t = 4; // t in f_1, hides t in f
        { // un-named block scope f_1_1
            int p;
            p = 5; // p in f_1_1, hides p in f_1
        }
        q = p; // q in f_1, p in f_1
    }
    return u = t; // u in f, t in f
}
```

```
f: // function scope f
  // t in f, x in f
  t = x
  // p in f_1, a in global
  p@f_1 = a@glb
  // t in f_1, hides t in f
  t@f_1 = 4
  // p in f_1_1, hides p in f_1
  p@f_1_1 = 5
  // q in f_1, p in f_1
  q@f_1 = p@f_1
  // u in f, t in f
  u = t
```

ST.glb: ST.glb.parent = null

a	int	global	4	0	null
f	int → int				
		func	0	0	ST.f

ST.f(): ST.f.parent = ST.glb

x	int	param	4	0	null
t	int	local	4	4	null
u	int	local	4	8	null
f_1	null	block	–		ST.f_1

ST.f_1: ST.f_1.parent = ST.f

p	int	local	4	0	null
q	int	local	4	4	null
t	int	local	4	8	null
f_1_1	null	block	–		ST.f_1_1

ST.f_1_1: ST.f_1_1.parent = ST.f_1

p	int	local	4	0	null

Columns: Name, Type, Category, Size, Offset, & Symtab

Grammar and Parsing for this example is discussed with the Parse Tree in 3-Address Code Generation

Nested Blocks Flattened

```
f: // function scope f
  // t in f, x in f
  t = x
  // p in f_1, a in global
  p@f_1 = a@glb
  // t in f_1, hides t in f
  t@f_1 = 4
  // p in f_1_1, hides p in f_1
  p@f_1_1 = 5
  // q in f_1, p in f_1
  q@f_1 = p@f_1
  // u in f, t in f
  u = t
```

```
f: // function scope f
  // t in f, x in f
  t = x
  // p in f_1, a in global
  p#1 = a@glb  // p@f_1
  // t in f_1, hides t in f
  t#3 = 4      // t@f_1
  // p in f_1_1, hides p in f_1
  p#4 = 5      // p@f_1_1
  // q in f_1, p in f_1
  q#2 = p#1    // q@f_1, p@f_1
  // u in f, t in f
  u = t
```

ST.f(): ST.f.parent = ST.glb

x	int	param	4	0	null
t	int	local	4	4	null
u	int	local	4	8	null
f_1	null	block	–		ST.f_1

ST.f_1: ST.f_1.parent = ST.f

p	int	local	4	0	null
q	int	local	4	4	null
t	int	local	4	8	null
f_1_1	null	block	–		ST.f_1_1

ST.f_1_1: ST.f_1_1.parent = ST.f_1

p	int	local	4	0	null

Columns: Name, Type, Category, Size, Offset, & Symtab

ST.f(): ST.f.parent = ST.glb

x	int	param	4	0	null
t	int	local	4	4	null
u	int	local	4	8	null
p#1	int	blk-local	4	0	null
q#2	int	blk-local	4	4	null
t#3	int	blk-local	4	8	null
p#4	int	blk-local	4	0	null

Example : Global & Function Scope: main() & add(): Source & TAC

```
int x, ar[2][3], y;
int add(int x, int y);
double a, b;
int add(int x, int y) {
    int t;
    t = x + y;
    return t;
}
void main() {
    int c;
    x = 1;
    y = ar[x][x];
    c = add(x, y);
    return;
}
```

```
add:    t#1 = x + y
        t = t#1
        return t

main:   t#1 = 1
        x = t#1
        t#2 = x * 12
        t#3 = x * 4
        t#4 = t#2 + t#3
        y = ar[t#4]
        param x
        param y
        c = call add, 2
        return
```

ST.glb: ST.glb.parent = null

x	int	global	4	0	null
ar	array(2, array(3, int))				
		global	24	4	null
y	int	global	4	28	null
add	int × int → int				
		func	0	32	ST.add()
a	double	global	8	32	null
b	double	global	8	40	null
main	void → void				
		func	0	48	ST.main()

ST.add(): ST.add.parent = ST.glb

y	int	param	4	0
x	int	param	4	4
t	int	local	4	8
t#1	int	temp	4	12

ST.main(): ST.main.parent = ST.glb

c	int	local	4	0
t#1	int	temp	4	4
t#2	int	temp	4	8
t#3	int	temp	4	12
t#4	int	temp	4	16

Columns: Name, Type, Category, Size, Offset, & Symtab

Grammar and Parsing for this example is discussed with the Parse Tree in 3-Address Code Generation

Example:Global, Extern & Local Static Data

```
// File Main.c
extern int n;
int Sum(int x) {
    static int lclStcSum = 0;

    lclStcSum += x;
    return lclStcSum;
}
int sum = -1;
void main() {
    int a = n;

    Sum(a);
    a *= a;
    sum = Sum(a);
    return;
}

// File Global.c
int n = 5;
```

```
    lclStcSum = 0
Sum:  lclStcSum = lclStcSum + x
    return lclStcSum

        sum = -1
main:   a = glb_n
        param a
        call Sum, 1
        a = a * a
        param a
        sum = call Sum, 1
        return
```

ST.glb (Main.c)

n	int	extern	4	0
Sum	int → int	func	0	4
sum	int	global	4	0
main	void → void	func	0	8

ST.glb (Global.c)

n	int	global	4	0

ST.Sum()

x	int	param	4	0
lclStcSum	int	static	4	4

ST.main()

a	int	local	4	0

Columns are: Name, Type, Category, Size, & Offset

Example: Binary Search

```
int bs(int a[], int l,
       int r, int v) {
    while (l <= r) {
        int m = (l + r) / 2;
        if (a[m] == v)
            return m;
        else
            if (a[m] > v)
                r = m - 1;
            else
                l = m + 1;
    }
    return -1;
}
```

```
100: if l < = r goto 102
101: goto 121
102: t1 = l + r
103: t2 = t1 / 2
104: m = t2
105: t3 = m * 4
106: t4 = a[t3]
107: if t4 == v goto 109
108: goto 111
109: return m
110: goto 100
```

```
111: t5 = m * 4
112: t6 = a[t5]
113: if t6 > v goto 115
114: goto 118
115: t7 = m - 1
116: r = t7
117: goto 100
118: t8 = m + 1
119: l = t8
120: goto 100
121: t9 = -1
122: return t9
```

ST.glb

bs	array(*, int) × int × int× int → int		
	func	0	0

Columns: Name, Type, Category, Size, & Offset

Temporary variables are numbered in the function scope – the effect of the respective block scope in the numbering is not considered. Hence, we show only a flattened symbol table

ST.bs()

a	array(*, int)	param	4	+16
l	int	param	4	+12
r	int	param	4	+8
r	int	param	4	+4
m	int	local	4	0
t1	int	temp	4	−4
t2	int	temp	4	−8
t3	int	temp	4	−12
t4	int	temp	4	−16
t5	int	temp	4	−20
t6	int	temp	4	−24
t7	int	temp	4	−28
t8	int	temp	4	−32
t9	int	temp	4	−36

Convention is opposite, reverse it.

Example: Transpose

```
int main() {
    int a[3][3];
    int i, j;
    for (i = 0; i < 3; ++i) {
        for (j = 0; j < i; ++j) {
            int t;
            t = a[i][j];
            a[i][j] = a[j][i];
            a[j][i] = t;
        }
    }
    return;
}
```

```
100: t01 = 0
101: i = t01
102: t02 = 3
103: if i < t02 goto 108
104: goto 134
105: t03 = i + 1
106: i = t03
107: goto 103
108: t04 = 0
109: j = t04
110: if j < i goto 115
111: goto 105
112: t05 = j + 1
113: j = t05
114: goto 110
115: t06 = 12 * i
116: t07 = 4 * j
117: t08 = t06 + t07
```

```
118: t09 = a[t08]
119: t = t09
120: t10 = 12 * i
121: t11 = 4 * j
122: t12 = t10 + t11
123: t13 = 12 * j
124: t14 = 4 * i
125: t15 = t13 + t14
126: t16 = a[t15]
127: a[t12] = t16
128: t17 = 12 * j
129: t18 = 4 * i
130: t19 = t17 + t18
131: a[t19] = t
132: goto 112
133: goto 105
134: return
```

ST.glb

main	void → void	func

ST.main()

a	array(3, array(3, int))			
		param	4	0
i	int	local	4	−4
j	int	local	4	−8
t01	int	temp	4	−12
t02	int	temp	4	−16
t03	int	temp	4	−20
t04	int	temp	4	−24
t05	int	temp	4	−28
t06	int	temp	4	−32
t07	int	temp	4	−36

ST.main()

t08	int	temp	4	−40
t09	int	temp	4	−44
t10	int	temp	4	−48
t11	int	temp	4	−52
t12	int	temp	4	−56
t13	int	temp	4	−60
t14	int	temp	4	−64
t15	int	temp	4	−68
t16	int	temp	4	−72
t17	int	temp	4	−76
t18	int	temp	4	−80
t19	int	temp	4	−84