Module 05: CS31003: Compilers: Machine Independent Translation

Pralay Mitra
Partha P Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

pralay@cse.iitkgp.ac.in
ppd@cse.iitkgp.ac.in

August 19, 20, 26 & 31 and
September 02, 03, 09 & 30, 2019

Module Objectives

- Understand Intermediate Representations
- Symbol Tables
- Understand Syntax Directed Translation
- Understand how Semantic Actions be guided by Syntactic Translation (using Attributed Grammars)

Module Outline

- Three Address Codes
- Symbol Table
  - Notion and Purpose
  - Scope Management Examples
  - Interface
  - Implementation
- Syntax-Directed Translation to Intermediate Codes for:
  - Arithmetic Expressions (and simple assignment)
  - Boolean Expressions (and elementary control flow)
  - Control Constructs (if, if-else, while, do-while, for, switch)
  - Variable declarations and datatypes
  - Translation by type
  - Arrays in Expressions
  - Type Expressions
  - Functions (definition, invocation, and computations)
  - Scope Management (nested lexical scopes)

Three Address Code

Intermediate Representations (IR)

- Each compiler uses 2-3 IRs
- Multi-Level Intermediate Representations
    - High-Level Representations (HIR)
        - Preserves loop structure and array bounds
        - Abstract Syntax Tree (AST) / DAG
            - Condensed form of parse tree
            - Useful for representing language constructs
            - Depicts the natural hierarchical structure of the source program
                * Each internal node represents an operator
                * Children of the nodes represent operands
                * Leaf nodes represent operands
            - DAG is more compact than AST because common sub expressions are eliminated
    - Mid-Level Representations (MIR):
        - Reflects range of features in a set of source languages
        - Language independent
        - Good for code generation for a number of architectures
        - Appropriate for most optimization opportunities
        - Three-Address Code (TAC)
    - Low-Level Representations (LIR):
        - Corresponds one to one to target machine instructions
        - Assembly Language of x86

Alternate Intermediate Representations

- SSA: Single Static Assignment
- RTL: Register transfer language
- Stack machines: P-code
- CFG: Control Flow Graph
- Dominator Trees
- DJ-graph: dominator tree augmented with join edges
- PDG: Program Dependence Graph
- VDG: Value Dependence Graph
- GURRR: Global unified resource requirement representation. Combines PDG with resource requirements
- Java intermediate bytecodes
- ...

Three Address Code

- Concepts
  - Address
  - Instruction

  In general these could be classes, specializing for every specific type.
- Uses only up to 3 addresses in every instruction
- Every 3 address instruction is represented by a quad – opcode, argument 1, argument 2, and result

Three Address Code

- Address Types
  - *Name*:
    Source program names appear as addresses in 3-Address Codes.
  - *Constant*:
    Many different types and their (implicit) conversions are allowed as deemed addresses.
  - *Compiler-Generated Temporary*:
    Create a distinct name each time a temporary is needed - good for optimization.

Three Address Code

- Instruction Types
  For Addresses `x`, `y`, `z`, and Label `L`
  - *Binary Assignment Instruction*: For a binary op (including arithmetic, logical, or bit operators):

    ```
    x = y op z
    ```
  - *Unary Assignment Instruction*: For a unary operator op (including unary minus, logical negation, shift operators, conversion operators):

    ```
    x = op y
    ```
  - *Copy Assignment Instruction*:

    ```
    x = y
    ```

Three Address Code

- Instruction Types
  For Addresses `x`, `y`, and Label `L`
  - *Unconditional Jump*:

    `goto L`
  - *Conditional Jump*:
    - *Value-based*:

      ```
      if x goto L
      ifFalse x goto L
      ```
    - *Comparison-based*: For a relational operator op (including $<$, $>$, $==$, $!=$, $\leq$, $\geq$):

      ```
      if x relop y goto L
      ```

Three Address Code

- Instruction Types
  For Addresses `p`, `x1`, `x2`, and `xN`
  - Procedure Call: A procedure call `p(x1, x2, ..., xN)` having $N \geq 0$ parameters is coded as:

    ```
    param x1
    param x2
    ...
    param xN
    y = call p, N
    ```

    Note that `N` is not redundant as procedure calls can be nested.
  - Return Value: Returning a return value and /or assigning it is optional. If there is a return value it is returned from the procedure `p` as:

    ```
    return n
    ```

Three Address Code

- Instruction Types
  For Addresses `x`, `y`, and `i`
  - *Indexed Copy Instructions*:

    ```
    x = y[i]
    x[i] = y
    ```
  - *Address and Pointer Assignment Instructions*:

    ```
    x = &y
    x = *y
    *x = y
    ```

Three Address Code

- Example

```
do i = i + 1; while (a[i] < v);
```

translates to

```
L: t1 = i + 1
   i = t1
   t2 = i * 8
   t3 = a[t2]
   if t3 < v goto L
```

The symbolic label is then given positional numbers as:

```
100: t1 = i + 1
101: i = t1
102: t2 = i * 8
103: t3 = a[t2]
104: if t3 < v goto 100
```

Three Address Code

- For

```
L: t1 = i + 1
   i = t1
   t2 = i * 8
   t3 = a[t2]
   if t3 < v goto L
```

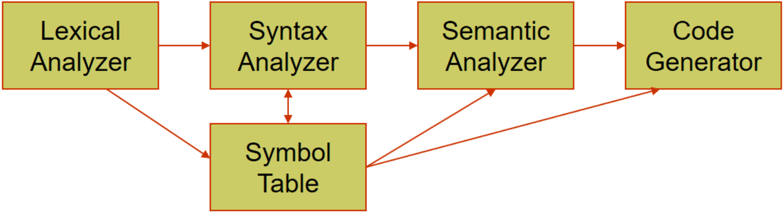
quads are represented as:

	op	**arg 1**	**arg 2**	**result**
0	$+$	i	1	t1
1	$=$	t1	null	i
2	*	i	8	t2
3	$=[]$	a	t2	t3
4	$<$	t3	v	L

Symbol Table

Symbol Table

- When identifiers are found by the lexical analyzer, they are entered into a **Symbol Table**, which will hold all relevant information about identifiers.
- This information is updated later by Syntax Analyzer, and used & updated even later by the Semantic Analyzer and the Code Generator.



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)
  - 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() {
    int a = b + 5;
    return a;
}

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

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

    return 0;
}
```

Dynamic Scoping

```
const int b = 5;

int foo() {
    int a = b + 5;
    return a;
}

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

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

    return 0;
}
```

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

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 constants, constant 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

Arithmetic Expressions

A Calculator Grammar

$$
\begin{array}{rlcl}
1: & L & \rightarrow & L\ S\ \backslash\mathrm{n} \\
2: & L & \rightarrow & S\ \backslash\mathrm{n} \\
3: & S & \rightarrow & \mathbf{id} = E \\
4: & E & \rightarrow & E + E \\
5: & E & \rightarrow & E - E \\
6: & E & \rightarrow & E * E \\
7: & E & \rightarrow & E\ /\ E \\
8: & E & \rightarrow & (E) \\
9: & E & \rightarrow & -\ E \\
10: & E & \rightarrow & \mathbf{num} \\
11: & E & \rightarrow & \mathbf{id}
\end{array}
$$

Attributes for Expression

$E.loc$: – Location to store the value of the expression.
– This will exist in the Symbol Table.

id.*loc*: – Location to store the value of the identifier **id**.
– This will exist in the Symbol Table.

num.*val*: – Value of the numeric (integer) constant.

Auxiliary Methods for Translation

gentemp(): – Generates a new temporary and inserts it in the Symbol Table
– Returns a pointer to the new entry in the Symbol Table

$emit(result, arg1, op, arg2)$:

– Spits a 3 Address Code of the form:

```
result = arg1 op arg2
```

– op usually is a binary operator. If `arg2` is missing, op is unary. If op also is missing, this is a copy instruction.

Expression Grammar with Actions

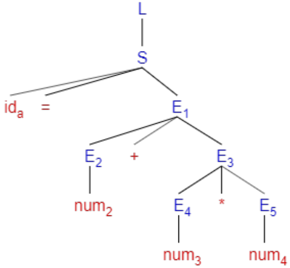
1:	L	$\rightarrow$	$L\ S$ \n	$\{\ \}$
2:	L	$\rightarrow$	S \n	$\{\ \}$
3:	S	$\rightarrow$	$\textbf{id} = E$	$\{\ emit(\textbf{id}.loc = E.loc);\ \}$ // No new temporary, copy code
4:	E	$\rightarrow$	$E_1 + E_2$	$\{\ E.loc = gentemp();$ $emit(E.loc = E_1.loc + E_2.loc);\ \}$
5:	E	$\rightarrow$	$E_1 - E_2$	$\{\ E.loc = gentemp();$ $emit(E.loc = E_1.loc - E_2.loc);\ \}$
6:	E	$\rightarrow$	$E_1 * E_2$	$\{\ E.loc = gentemp();$ $emit(E.loc = E_1.loc * E_2.loc);\ \}$
7:	E	$\rightarrow$	$E_1\ /\ E_2$	$\{\ E.loc = gentemp();$ $emit(E.loc = E_1.loc / E_2.loc);\ \}$
8:	E	$\rightarrow$	(E_1)	$\{\ E.loc = E_1.loc;\ \}$ // No new temporary, no code
9:	E	$\rightarrow$	$-\ E_1$	$\{\ E.loc = gentemp();$ $emit(E.loc = -E_1.loc);\ \}$
10:	E	$\rightarrow$	$\textbf{num}$	$\{\ E.loc = gentemp();$ $emit(E.loc = \textbf{num}.val);\ \}$
11:	E	$\rightarrow$	$\textbf{id}$	$\{\ E.loc = \textbf{id}.loc;\ \}$ // No new temporary, no code

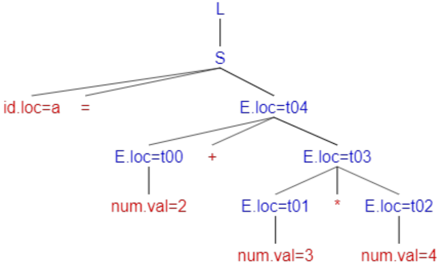
Intermediate 3 address codes are emitted as soon as they are formed.

Translation Example

```
$ ./a.out
a = 2 + 3 * 4
    t00 = 2
    t01 = 3
    t02 = 4
    t03 = t01 * t02
    t04 = t00 + t03
    a = t04
$
$
```

Reductions	**TAC**
$E \rightarrow$ **num**	`t00 = 2`
$E \rightarrow$ **num**	`t01 = 3`
$E \rightarrow$ **num**	`t02 = 4`
$E \rightarrow E_1 * E_2$	`t03 = t01 * t02`
$E \rightarrow E_1 + E_2$	`t04 = t00 + t03`
$S \rightarrow$ **id** $=$ E	`a = t04`





Yacc Specs (calc.y) for Calculator Grammar

```
%{
#include <string.h>
#include <iostream>
#include "parser.h"
extern int yylex();
void yyerror(const char *s);
#define NSYMS 20 /* max # of symbols */
symboltable symtab[NSYMS];
%}

%union {
    int intval;
    struct symtab *symp;
}

%token <symp> NAME
%token <intval> NUMBER

%left ’+’ ’-’
%left ’*’ ’/’
%nonassoc UMINUS

%type <symp> expression
%%

stmt_list: statement ’\n’
         | stmt_list statement ’\n’
         ;
```

```
statement: NAME ’=’ expression
    { emit($1->name, $3->name); }
         ;

expression: expression ’+’ expression
    { $$ = gentemp();
      emit($$->name, $1->name, ’+’, $3->name); }
          | expression ’-’ expression
    { $$ = gentemp();
      emit($$->name, $1->name, ’-’, $3->name); }
          | expression ’*’ expression
    { $$ = gentemp();
      emit($$->name, $1->name, ’*’, $3->name); }
          | expression ’/’ expression
    { $$ = gentemp();
      emit($$->name, $1->name, ’/’, $3->name); }
          | ’(’ expression ’)’
    { $$ = $2; }
          | ’-’ expression %prec UMINUS
    { $$ = gentemp();
      emit($$->name, $2->name, ’-’);  }
          | NAME { $$ = $1; }
          | NUMBER
    { $$ = gentemp();
      printf("\t%s = %d\n", $$->name, $1); }
          ;
%%
```

Yacc Specs (calc.y) for Calculator Grammar

```
/* Look-up Symbol Table */
symboltable *symlook(char *s) {
    char *p;
    struct symtab *sp;
    for(sp = symtab;
        sp < &symtab[NSYMS]; sp++) {
        /* is it already here? */
        if (sp->name &&
           !strcmp(sp->name, s))
            return sp;
        if (!sp->name) {
        /* is it free */
            sp->name = strdup(s);
            return sp;
        }
        /* otherwise continue to next */
    }
    yyerror("Too many symbols");
    exit(1); /* cannot continue */
} /* symlook */

/* Generate temporary variable */
symboltable *gentemp() {
    static int c = 0; /* Temp counter */
    char str[10]; /* Temp name */
    /* Generate temp name */
    sprintf(str, "t%02d", c++);
    /* Add temporary to symtab */
    return symlook(str);
}
```

```
/* Output 3-address codes */
void emit(char *s1,     // Result
          char *s2,     // Arg 1
          char c = 0,   // Operator
          char *s3 = 0) // Arg 2
{
    if (s3)
        /* Assignment with Binary operator */
        printf("\t%s = %s %c %s\n",s1, s2, c, s3);
    else
        if (c)
            /* Assignment with Unary operator */
            printf("\t%s = %c %s\n",s1, c, s2);
        else
            /* Simple Assignment */
            printf("\t%s = %s\n",s1, s2);
}

void yyerror(const char *s) {
    std::cout << s << std::endl;
}

int main() {
    yyparse();
}
```

Header (y.tab.h) for Calculator

```
/* A Bison parser, made by GNU Bison 2.5.  */
/* Tokens. */
#ifndef YYTOKENTYPE
# define YYTOKENTYPE
   /* Put the tokens into the symbol table, so that GDB and other debuggers know about them. */
   enum yytokentype {
     NAME = 258,
     NUMBER = 259,
     UMINUS = 260
   };
#endif
/* Tokens. */
#define NAME 258
#define NUMBER 259
#define UMINUS 260

#if ! defined YYSTYPE && ! defined YYSTYPE_IS_DECLARED
typedef union YYSTYPE {
#line 11 "calc.y" /* Line 2068 of yacc.c  */

    int intval;
    struct symtab *symp;

#line 67 "y.tab.h" /* Line 2068 of yacc.c  */
} YYSTYPE;
# define YYSTYPE_IS_TRIVIAL 1
# define yystype YYSTYPE /* obsolescent; will be withdrawn */
# define YYSTYPE_IS_DECLARED 1
#endif
```

Header (parser.h) for Calculator

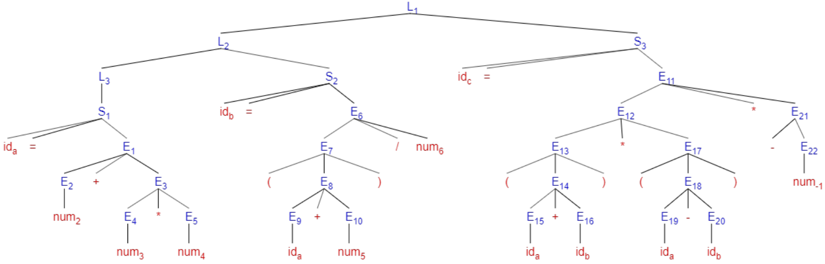
```
#ifndef __PARSER_H
#define __PARSER_H

/* Symbol Table Entry */
typedef struct symtab {
    char *name;
    int value;
} symboltable;

/* Look-up Symbol Table */
symboltable *symlook(char *);

/* Generate temporary variable */
symboltable *gentemp();

/* Output 3-address codes */
/* if s3 != 0 ==> Assignment with Binary operator */
/* if s3 == 0 && c != 0 ==> Assignment with Unary operator */
/* if s3 == 0 && c == 0 ==> Simple Assignment */
void emit(char *s1, char *s2, char c = 0, char *s3 = 0);

#endif // __PARSER_H
```

Flex Specs (calc.l) for Calculator Grammar

```
%{
#include <math.h>
#include "y.tab.h"
#include "parser.h"
%}

ID        [A-Za-z][A-Za-z0-9]*

%%
[0-9]+    {
            yylval.intval = atoi(yytext);
            return NUMBER;
          }

[ \t]     ;         /* ignore white space */


{ID}      { /* return symbol pointer */
            yylval.symp = symlook(yytext);
            return NAME;
          }

"$"          { return 0; /* end of input */ }

\n|.      return yytext[0];
%%
```

Sample Run

```
$ ./a.out
a = 2 + 3 * 4
    t00 = 2
    t01 = 3
    t02 = 4
    t03 = t01 * t02
    t04 = t00 + t03
    a = t04
b = (a + 5) / 6
    t05 = 5
    t06 = a + t05
    t07 = 6
    t08 = t06 / t07
    b = t08
c = (a + b) * (a - b) * -1
    t09 = a + b
    t10 = a - b
    t11 = t09 * t10
    t12 = 1
    t13 = - t12
    t14 = t11 * t13
    c = t14
$
$
```



Translation with Lazy Spitting

Intermediate 3 address codes are formed as quads and stored in an array. The quads are spit at the end to output. This can help optimization later.

Note on Yacc Specs (calc.y)

- `class quad` is used to represent a quad
- It has the following fields:

Name	Type	Remarks
`op`	`opcodeType`	Specifies the type of 3-address instruction. This can be binary operator, unary operator or copy
`arg1`	`char *`	First argument. If the actual argument is a numeric constant, we use decimal form as a string
`arg2`	`char *`	Second argument
`result`	`char *`	Result

Yacc Specs (calc.y) for Calculator Grammar

```
%{
#include <string.h>
#include <iostream>
#include "parser.h"
extern int yylex();
void yyerror(const char *s);
#define NSYMS 20    // max # of symbols
symboltable symtab[NSYMS];
quad *qArray[NSYMS]; // Store of Quads
int quadPtr = 0; // Index of next quad
%}

%union {
    int intval;
    struct symtab *symp;
}

%token <symp> NAME
%token <intval> NUMBER

%left ’+’ ’-’
%left ’*’ ’/’
%nonassoc UMINUS

%type <symp> expression
%%

start: statement_list
    { for(int i = 0; i < quadPtr; i++)
          qArray[i]->print(); }
     ;
```

```
statement_list:    statement ’\n’
              |    statement_list statement ’\n’
              ;
statement: NAME ’=’ expression
  { qArray[quadPtr++] =
    new quad(COPY, $1->name, $3->name); }
         ;
expression: expression ’+’ expression
  { $$ = gentemp(); qArray[quadPtr++] =
  new quad(PLUS, $$->name, $1->name, $3->name); }
          | expression ’-’ expression
  { $$ = gentemp(); qArray[quadPtr++] =
  new quad(MINUS, $$->name, $1->name, $3->name); }
          | expression ’*’ expression
  { $$ = gentemp(); qArray[quadPtr++] =
  new quad(MULT, $$->name, $1->name, $3->name); }
          | expression ’/’ expression
  { $$ = gentemp(); qArray[quadPtr++] =
  new quad(DIV, $$->name, $1->name, $3->name); }
          | ’(’ expression ’)’      { $$ = $2; }
          | ’-’ expression %prec UMINUS
  { $$ = gentemp(); qArray[quadPtr++] =
  new quad(UNARYMINUS, $$->name, $2->name); }
          | NAME                    { $$ = $1; }
          | NUMBER
  { $$ = gentemp(); qArray[quadPtr++] =
  new quad(COPY, $$->name, $1); }
          ;
%%
```

Yacc Specs (calc.y) for Calculator Grammar

```
/* Look-up Symbol Table */
symboltable *symlook(char *s) {
    char *p;
    struct symtab *sp;
    for(sp = symtab;
        sp < &symtab[NSYMS]; sp++) {
        /* is it already here? */
        if (sp->name &&
           !strcmp(sp->name, s))
            return sp;
        if (!sp->name) {
        /* is it free */
            sp->name = strdup(s);
            return sp;
        }
        /* otherwise continue to next */
    }
    yyerror("Too many symbols");
    exit(1); /* cannot continue */
} /* symlook */

/* Generate temporary variable */
symboltable *gentemp() {
    static int c = 0; /* Temp counter */
    char str[10]; /* Temp name */
    /* Generate temp name */
    sprintf(str, "t%02d", c++);
    /* Add temporary to symtab */
    return symlook(str);
}
```

```
void yyerror(const char *s) {
    std::cout << s << std::endl;
}

int main() {
    yyparse();
}
```

Header (y.tab.h) for Calculator

```
/* A Bison parser, made by GNU Bison 2.5.  */
/* Tokens. */
#ifndef YYTOKENTYPE
# define YYTOKENTYPE
   /* Put the tokens into the symbol table, so that GDB and other debuggers know about them. */
   enum yytokentype {
     NAME = 258,
     NUMBER = 259,
     UMINUS = 260
   };
#endif
/* Tokens. */
#define NAME 258
#define NUMBER 259
#define UMINUS 260

#if ! defined YYSTYPE && ! defined YYSTYPE_IS_DECLARED
typedef union YYSTYPE {
#line 13 "calc.y" /* Line 2068 of yacc.c  */

    int intval;
    struct symtab *symp;

#line 67 "y.tab.h" /* Line 2068 of yacc.c  */
} YYSTYPE;
# define YYSTYPE_IS_TRIVIAL 1
# define yystype YYSTYPE /* obsolescent; will be withdrawn */
# define YYSTYPE_IS_DECLARED 1
#endif
```

Header (parser.h) for Calculator

```
#ifndef __PARSER_H
#define __PARSER_H

#include<stdio.h>

/* Symbol Table Entry */
typedef struct symtab {
    char *name;
    int value;
}symboltable;

/* Look-up Symbol Table */
symboltable *symlook(char *);

/* Generate temporary variable */
symboltable *gentemp();

typedef enum {
    PLUS = 1,
    MINUS,
    MULT,
    DIV,
    UNARYMINUS,
    COPY,
} opcodeType;
```

```
class quad {
    opcodeType op;
    char *result, *arg1, *arg2;
public:
    quad(opcodeType op1, char *s1, char *s2, char *s3=0):
        op(op1), result(s1), arg1(s2), arg2(s3) { }
    quad(opcodeType op1, char *s, int num):
        op(op1), result(s1), arg1(0), arg2(0)
    {
        arg1 = new char[15];
        sprintf(arg1, "%d", num);
    }
    void print() {
        if ((op <= DIV) && (op >= PLUS)) { // Binary Op
            printf("%s = %s ",result, arg1);
            switch (op) {
                case PLUS: printf("+"); break;
                case MINUS: printf("-"); break;
                case MULT: printf("*"); break;
                case DIV: printf("/"); break;
            }
            printf(" %s\n",arg2);
        }
        else
            if (op == UNARYMINUS) // Unary Op
                printf("%s = - %s\n",result, arg1);
            else // Copy
                printf("%s = %s\n",result, arg1);
    }
};
#endif // __PARSER_H
```

Flex Specs (calc.l) for Calculator Grammar

```
%{
#include <math.h>
#include "y.tab.h"
#include "parser.h"
%}

ID        [A-Za-z][A-Za-z0-9]*

%%
[0-9]+    {
            yylval.intval = atoi(yytext);
            return NUMBER;
          }

[ \t]     ;         /* ignore white space */


{ID}      { /* return symbol pointer */
            yylval.symp = symlook(yytext);
            return NAME;
          }

"$"          { return 0; /* end of input */ }

\n|.      return yytext[0];
%%
```

Sample Run

Output

```
$ ./a.out
a = 2 + 3 * 4
b = (a + 5) / 6
c = (a + b) * (a - b) * -1
    t00 = 2
    t01 = 3
    t02 = 4
    t03 = t01 * t02
    t04 = t00 + t03
    a = t04
    t05 = 5
    t06 = a + t05
    t07 = 6
    t08 = t06 / t07
    b = t08
    t09 = a + b
    t10 = a - b
    t11 = t09 * t10
    t12 = 1
    t13 = - t12
    t14 = t11 * t13
    c = t14
$
```

Boolean Expressions

Boolean Expression Grammar

$$\begin{array}{llll}
1: & B & \rightarrow & B_1 \mid\mid B_2 \\
2: & B & \rightarrow & B_1 \;\&\&\; B_2 \\
3: & B & \rightarrow & !B_1 \\
4: & B & \rightarrow & (B_1) \\
5: & B & \rightarrow & E_1 \;\texttt{relop}\; E_2 \\
6: & B & \rightarrow & \texttt{true} \\
7: & B & \rightarrow & \texttt{false}
\end{array}$$

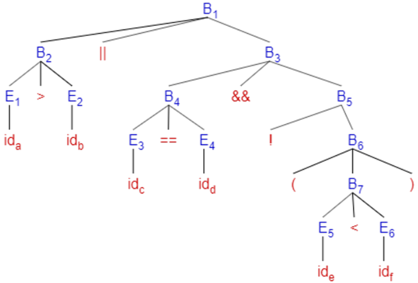
`relop` is any one of:

$<, <=, >, >=, ==, !=$

Boolean Expression Example: Translation by Value

a > b || c == d && !(e < f)

```
100: t1 = a > b
101: t2 = c == d
102: t3 = e < f
103: t4 = !t3
104: t5 = t3 && t4
105: t6 = t1 || t5
```

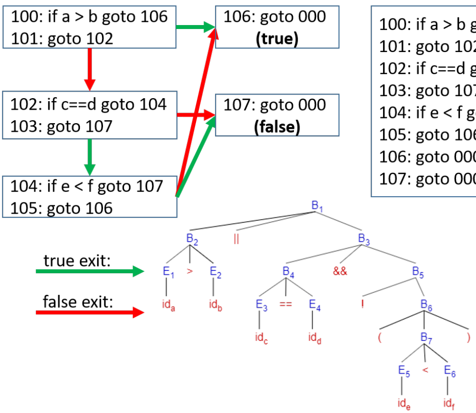


Translation by Value:

- May not be very useful, as Boolean values are typically used for control flow
- May not use short-cut of computation

Boolean Expression Example: Translation by Control Flow

a > b || c == d && !(e < f)



100: if a > b goto 106
101: goto 102

106: goto 000
(true)

102: if c==d goto 104
103: goto 107

107: goto 000
(false)

104: if e < f goto 107
105: goto 106

true exit:

false exit:

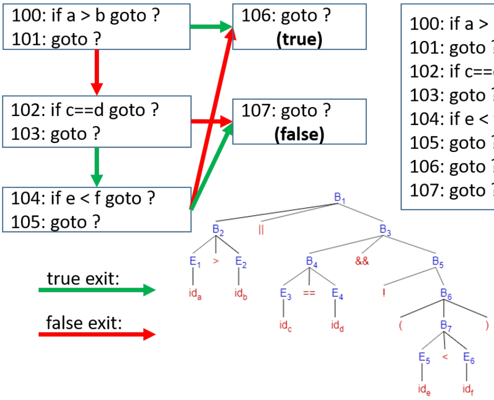
100: if a > b goto 106
101: goto 102
102: if c==d goto 104
103: goto 107
104: if e < f goto 107
105: goto 106
106: goto 000 **(true)**
107: goto 000 **(false)**

Translation by Control:

- Useful for control flow
- Uses short-cut of computation

Boolean Expression Example: Translation by Control Flow

a > b | | c == d && !(e < f)



```
100: if a > b goto ?
101: goto ?
```

```
102: if c==d goto ?
103: goto ?
```

```
104: if e < f goto ?
105: goto ?
```

```
106: goto ?
     (true)
```

```
107: goto ?
     (false)
```

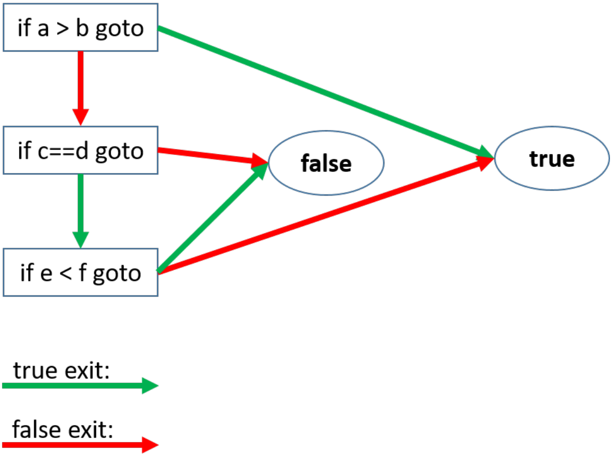
```
100: if a > b goto ?
101: goto ?
102: if c==d goto ?
103: goto ?
104: if e < f goto ?
105: goto ?
106: goto ? (true)
107: goto ? (false)
```

Translation by Control:

- How to get the target address of goto's?
- Can we optimize goto to goto's / fall-through's

Boolean Expression Example: Translation by Control Flow: Abstracted

a > b || c == d && !(e < f)



if a > b goto

if c==d goto

if e < f goto

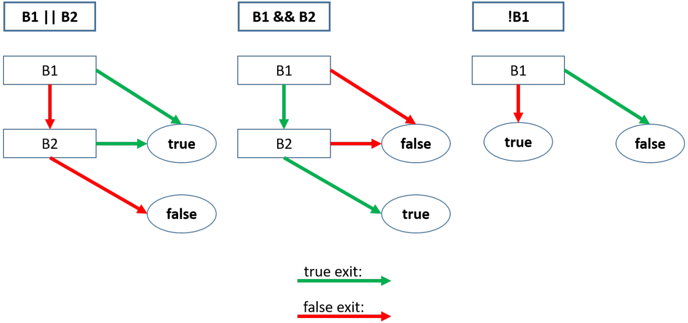
false

true

true exit:

false exit:

Boolean Expression: Scheme of Translation by Control Flow



Attributes / Global for Boolean Expression

$B.truelist$: – List of (indices of) quads having dangling **true exits** for the Boolean expression.

$B.falselist$: – List of (indices of) quads having dangling **false exits** for the Boolean expression.

$B.loc$: – Location to store the value of the Boolean expression (optional).

$nextinstr$: – Global counter to the array of quads – the index of the next quad to be generated.

$M.instr$: – Index of the quad generated at M.

Auxiliary Methods for Back-patching

$makelist(i)$: – Creates a new list containing only i, an index into the array of quad's.
– Returns a pointer to the newly created list

$merge(p_1, p_2)$: – Concatenates the lists pointed to by p_1 and p_2.
– Returns a pointer to the concatenated list

$backpatch(p, i)$: – Inserts i as the target label for each of the quads on the list pointed to by p.

Back-patching Boolean Expression Grammar

1: $B \rightarrow B_1 \mid\mid M\ B_2$

2: $B \rightarrow B_1\ \&\&\ M\ B_2$

3: $B \rightarrow\ !B_1$

4: $B \rightarrow (B_1)$

5: $B \rightarrow E_1\ \texttt{relop}\ E_2$

6: $B \rightarrow \texttt{true}$

7: $B \rightarrow \texttt{false}$

8: $M \rightarrow \epsilon$ // Marker rule

Back-patching Boolean Expression Grammar with Actions

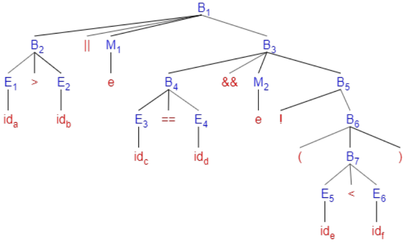
1:	B	$\rightarrow$	$B_1 \mid\mid M\ B_2$	
				$\{\ backpatch(B_1.falselist, M.instr);$ $B.truelist = merge(B_1.truelist, B_2.truelist);$ $B.falselist = B_2.falselist;\ \}$
2:	B	$\rightarrow$	$B_1\ \&\&\ M\ B_2$	
				$\{\ backpatch(B_1.truelist, M.instr);$ $B.truelist = B_2.truelist;$ $B.falselist = merge(B_1.falselist, B_2.falselist);\ \}$
3:	B	$\rightarrow$	$!B_1$	$\{\ B.truelist = B_1.falselist;$ $B.falselist = B_1.truelist;\ \}$
4:	B	$\rightarrow$	(B_1)	$\{\ B.truelist = B_1.truelist;$ $B.falselist = B_1.falselist;\ \}$
5:	B	$\rightarrow$	$E_1\ \texttt{relop}\ E_2$	
				$\{\ B.truelist = makelist(nextinstr);$ $B.falselist = makelist(nextinstr + 1);$ $emit(\text{"}if\text{"}, E_1.loc,\ relop.op,\ E_2.loc,\ \text{"}goto\text{"},\ \text{"}.....\text{"});\ \}$ $emit(\text{"}goto\text{"},\ \text{"}.....\text{"});\ \}$
6:	B	$\rightarrow$	$\texttt{true}$	$\{\ B.truelist = makelist(nextinstr);$ $emit(\text{"}goto\text{"},\ \text{"}.....\text{"});\ \}$
7:	B	$\rightarrow$	$\texttt{false}$	$\{\ B.falselist = makelist(nextinstr);$ $emit(\text{"}goto\text{"},\ \text{"}.....\text{"});\ \}$
8:	M	$\rightarrow$	ϵ	$\{\ M.instr = nextinstr;\ \}$

Back-patching Boolean Expression Grammar with Actions – Home Assignment

```
9:   $B \rightarrow B_1 \,\hat{}\, M\ B_2$
        $\{\ backpatch(B_1.truelist, nextinstr);$
          $emit(B_1.loc, "=", true);$
          $emit("goto",\ M.instr);$
          $backpatch(B_1.falselist, nextinstr);$
          $emit(B_1.loc, "=", false);$
          $emit("goto",\ M.instr);$

          $B.truelist = makelist(nextinstr);$
          $backpatch(B_2.falselist, nextinstr);$
          $emit("if", B_1.loc,\ "goto",\ ".....");$
          $B.falselist = makelist(nextinstr);$
          $emit("goto",\ ".....");$

          $temp = makelist(nextinstr);$
          $B.falselist = merge(B.falselist, temp);$
          $backpatch(B_2.truelist, nextinstr);$
          $emit("if", B_1.loc,\ "goto",\ ".....");$
          $temp = makelist(nextinstr);$
          $B.truelist = merge(B.truelist, temp);$
          $emit("goto",\ "....."); \}$
```

Example: Boolean Expression

```
a > b || c == d && !(e < f)
```



Order of Reductions

Seq. #: (Prod. #)	**Production**		
1:(5)	B_2	$\rightarrow$	E_1 `relop` E_2
2:(8)	M_1	$\rightarrow$	ϵ
3:(5)	B_4	$\rightarrow$	E_3 `relop` E_4
4:(8)	M_2	$\rightarrow$	ϵ
5:(5)	B_7	$\rightarrow$	E_5 `relop` E_6
6:(4)	B_6	$\rightarrow$	(B_7)
7:(3)	B_5	$\rightarrow$	$!B_6$
8:(2)	B_3	$\rightarrow$	B_4 && M_2 B_5
9:(1)	B_1	$\rightarrow$	B_2 || M_1 B_3

```
[1] 100: if a > b goto ?
[1] 101: goto 102           // [8] BP(B2.FL, M1.I)
[3] 102: if c == d goto 104 // [9] BP(B4.TL, M2.I)
[3] 103: goto ?
[5] 104: if e < f goto ?
[5] 105: goto ?

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

[1] B2.TL = {100}
[1] B2.FL = {101}
[2] M1.I  = 102
[3] B4.TL = {102}
[3] B4.FL = {103}
[4] M2.I  = 104
[5] B7.TL = {104}
[5] B7.FL = {105}
[6] B6.TL = B7.TL = {104}
[6] B6.FL = B7.FL = {105}
[7] B5.TL = B6.FL = {105}
[7] B5.FL = B6.TL = {104}
[8] B3.TL = B5.TL = {105}
[8] B3.FL = B4.FL U B5.FL = {103, 104}
[9] B1.TL = B2.TL U B3.TL = {100, 105}
[9] B1.FL = B3.FL = {103, 104}

---------------------------
[#] Reduction Sequence #
```

Handling Control Constructs

Control Constructs

Control Construct Grammar

$$
\begin{array}{llll}
1: & S & \rightarrow & \{\ L\ \} \\
2: & S & \rightarrow & \textbf{id} = E\ \textbf{;} \\
3: & S & \rightarrow & \textbf{if}\ (B)\ S \\
4: & S & \rightarrow & \textbf{if}\ (B)\ S\ \textbf{else}\ S \\
5: & S & \rightarrow & \textbf{while}\ (B)\ S \\
6: & L & \rightarrow & L\ S \\
7: & L & \rightarrow & S
\end{array}
$$

Attributes for Control Construct

$S.nextlist$: – List of (indices of) quads having dangling **exit**s for statement S.

$L.nextlist$: – List of (indices of) quads having dangling **exit**s for (list of) statements L.

Back-patching Control Construct Grammar

1: $S \rightarrow \{\ L\ \}$
2: $S \rightarrow \textbf{id} = E\ \textbf{;}$
3: $S \rightarrow \textbf{if}\ (B)\ M\ S_1$
4: $S \rightarrow \textbf{if}\ (B)\ M_1\ S_1\ N\ \textbf{else}\ M_2\ S_2$
5: $S \rightarrow \textbf{while}\ M_1\ (B)\ M_2\ S_1$
6: $L \rightarrow L_1\ M\ S$
7: $L \rightarrow S$
8: $M \rightarrow \epsilon$ // Marker rule
9: $N \rightarrow \epsilon$ // Fall-through Guard rule

Back-patching Control Construct Grammar with Actions

1:	S	$\rightarrow$	$\{\ L\ \}$	$\{\ S.nextlist = L.nextlist;\ \}$
2:	S	$\rightarrow$	$\textbf{id} =\ E\ ;$	$\{\ S.nextlist = null;$ $emit(\textbf{id}.loc,\ ''='',\ E.loc);\ \}$
3:	S	$\rightarrow$	$\textbf{if}\ (B)\ M\ S_1$	$\{\ backpatch(B.truelist, M.instr);$ $S.nextlist = merge(B.falselist, S_1.nextlist);\ \}$
4:	S	$\rightarrow$	$\textbf{if}\ (B)\ M_1\ S_1\ N\ \textbf{else}\ M_2\ S_2$	$\{\ backpatch(B.truelist, M_1.instr);$ $backpatch(B.falselist, M_2.instr);$ $temp = merge(S_1.nextlist, N.nextlist);\ \}$ $S.nextlist = merge(temp, S_2.nextlist);\ \}$
5:	S	$\rightarrow$	$\textbf{while}\ M_1\ (B)\ M_2\ S_1$	$\{\ backpatch(S_1.nextlist, M_1.instr);$ $backpatch(B.truelist, M_2.instr);$ $S.nextlist = B.falselist;$ $emit(''goto'',\ M_1.instr);\ \}$
6:	L	$\rightarrow$	$L_1\ M\ S$	$\{\ backpatch(L_1.nextlist, M.instr);$ $L.nextlist = S.nextlist;\ \}$

Back-patching Control Construct Grammar with Actions – Class Assignment

$$
\begin{array}{llll}
7: & L \rightarrow & S & \{\ L.nextlist = S.nextlist;\ \} \\
8: & M \rightarrow & \epsilon & \{\ M.instr = nextinstr;\ \} \\
9: & N \rightarrow & \epsilon & \{\ N.nextlist = makelist(nextinstr); \\
 & & & \quad emit(''goto'', ''.....'');\ \} \\
10: & S \rightarrow & \multicolumn{2}{l}{\textbf{do}\ M_1\ S_1\ M_2\ \textbf{while}\ \textbf{(}\ B\ \textbf{);}} \\
 & & & \{\ backpatch(B.truelist, M_1.instr); \\
 & & & \quad backpatch(S_1.nextlist, M_2.instr); \\
 & & & \quad S.nextlist = B.falselist;\ \} \\
11: & S \rightarrow & \multicolumn{2}{l}{\textbf{for}\ \textbf{(}\ E_1\ \textbf{;}\ M_1\ B\ \textbf{;}\ M_2\ E_2\ N\ \textbf{)}\ M_3\ S_1} \\
 & & & \{\ backpatch(B.truelist, M_3.instr); \\
 & & & \quad backpatch(N.nextlist, M_1.instr); \\
 & & & \quad backpatch(S_1.nextlist, M_2.instr); \\
 & & & \quad emit(''goto''\ \ M_2.instr); \\
 & & & \quad S.nextlist = B.falselist;\ \} \\
12: & E \rightarrow & \textbf{id} & \{\ E.loc = \textbf{id}id.loc;\ \} \\
13: & E \rightarrow & \textbf{num} & \{\ E.loc = gentemp(); \\
 & & & \quad emit(E.loc, ''\ =\ '',\ num.val);\ \}
\end{array}
$$

Example: $S \rightarrow$ **if** (B) M_1 S_1 N **else** M_2 S_2

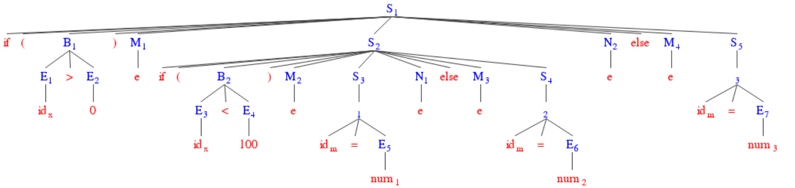
```
if (x > 0) if (x < 100) m = 1; else m = 2; else m = 3;
```

Order of Reductions

S#	Production
01:	$B_1 \rightarrow E_1$ `relop` E_2
02:	$M_1 \rightarrow \epsilon$
03:	$B_2 \rightarrow E_3$ `relop` E_4
04:	$M_2 \rightarrow \epsilon$
05:	$S_3 \rightarrow id_m = E_5$
06:	$N_1 \rightarrow \epsilon$
07:	$M_3 \rightarrow \epsilon$
08:	$S_4 \rightarrow id_m = E_6$
09:	$S_2 \rightarrow$ **if** (B_2) M_2 S_3 N_1 **else** M_3 S_4
10:	$N_2 \rightarrow \epsilon$
11:	$M_4 \rightarrow \epsilon$
12:	$S_5 \rightarrow id_m = E_7$
13:	$S_1 \rightarrow$ **if** (B_1) M_1 S_2 N_2 **else** M_4 S_5

```
[01] 100: if x > 0 goto 102   // [13] BP(B1.TL, M1.I)
[01] 101: goto 108            // [13] BP(B1.FL, M4.I)
[03] 102: if x < 100 goto 104 // [09] BP(B2.TL, M2.I)
[03] 103: goto 106            // [09] BP(B2.FL, M3.I)
[05] 104: m = 1
[06] 105: goto ___
[08] 106: m = 2
[10] 107: goto ___
[12] 108: m = 3
```

```
[01] B1.TL= {100}   [07] M3.I = 106
[01] B1.FL= {101}   [08] S4.NL= {}
[02] M1.I = 102     [09] S2.NL= S3.NL U N1.NL U S4.NL= {105}
[03] B2.TL= {102}   [10] N2.NL= {107}
[03] B2.FL= {103}   [11] M4.I = 108
[04] M2.I = 104     [12] S5.NL= {}
[05] S3.NL= {}      [13] S1.NL= S2.NL U N2.NL U S5.NL= {105,
[06] N1.NL= {105}                                      107}
```



Handling **goto**

Maintain a Label Table having the following information and lookup(Label) method:

- ID of Label – This will be entered to Label Table either when a label is defined or it is used as a target for a **goto** before being defined. So if this ID exists in the table, it has been encountered already
- ADDR, Address of Label (index of quad) – This is set from the definition of a label. Hence it will be null as long as a label has been encountered in one or more **goto**'s but not defined yet
- LST, List of dangling **goto**'s for this label – This will be null if ADDR is not null

```
L1: ...       // If L1 exists in Label Table
              //    if (ADDR = null)
              //        ADDR = nextinstr
              //        backpatch LST with ADDR
              //        LST = null
              //    else
              //        duplicate definition of label L1 - an error
              // If L1 does not exist, make an entry
              //    ADDR = nextinstr
              //    LST = null
```

Handling **goto**

```
goto L1;  // If L1 exists in Label Table
          //    if (ADDR = null) // Forward jump already seen
          //        LST = merge(LST, makelist(nextinstr));
          //    else // Target crossed - a backward jump
          //        use ADDR
          // If L1 does not exist, make an entry
          //    ADDR = null // New forward jump
          //    LST = makelist(nextinstr);
```

Back-patching Control Construct Grammar with Actions – Home Assignment

$S \rightarrow$ **switch (** E **)** S_1
$S \rightarrow$ **case num:** S_1
$S \rightarrow$ **default:** S_1

Using Mutually Exclusive "case" Clauses - Unlike C

Synthesized Attributes		Inherited Attributes	
	Code to Evaluate E into **t**		Code to Evaluate E into **t**
	goto test		**if t !=** V_1 **goto** L_1
L_1:	Code for S_1		Code for S_1
	goto next		**goto next**
L_2:	Code for S_2	L_1:	**if t !=** V_2 **goto** L_2
	goto next		Code for S_2
	...		**goto next**
L_{n-1}:	Code for S_{n-1}	L_2:	
	goto next		...
L_n:	Code for S_n	L_{n-2}:	**if t !=** V_{n-1} **goto** L_{n-1}
	goto next		Code for S_{n-1}
test:	**if t =** V_1 **goto** L_1		**goto next**
	if t = V_2 **goto** L_2	L_{n-1}:	Code for S_n
	...	**next**:	
	if t = V_{n-1} **goto** L_{n-1}		
	goto L_n		
next:			

Back-patching Control Construct Grammar with Actions – Home Assignment

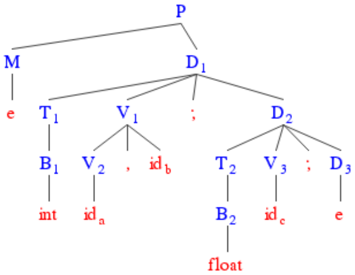
Design suitable schemes to translate **break** and **continue** statements:

$$
\begin{array}{lcl}
S & \rightarrow & \textbf{break;} \\
S & \rightarrow & \textbf{continue;}
\end{array}
$$

Types & Declarations

Declaration Grammar

$$\begin{array}{llll}
0: & P & \rightarrow & M\ D \\
1: & D & \rightarrow & T\ V\ ;\ D \\
2: & D & \rightarrow & \epsilon \\
3: & V & \rightarrow & V\ ,\ \textbf{id} \\
4: & V & \rightarrow & \textbf{id} \\
5: & T & \rightarrow & B \\
6: & B & \rightarrow & \textbf{int} \\
7: & B & \rightarrow & \textbf{float} \\
8: & M & \rightarrow & \epsilon
\end{array}$$



Example: `int a, b; float c;`

Name	Type	Size	Offset
a	int	4	0
b	int	4	4
c	float	8	8

Inherited Attribute

Consider the following attributes for types:

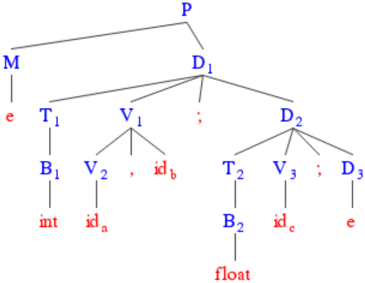
type: Type expression for B, T.

width: The width of a type (B, T), that is, the number of storage units (bytes) needed for objects of that type. It is integral for basic types.

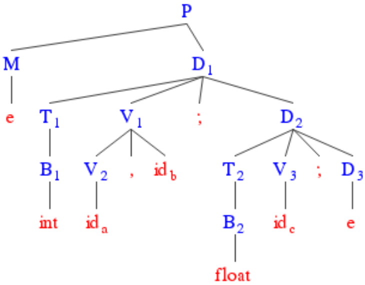
In the context of:

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

when $V \to$ **id** (or $V \to V$ **, id**) is reduced, we need to set the type (size) for **id** in the symbol table. However, the type (size) is not available from the children of V as *Synthesized Attributes*. Rather, it is available in T ($T.type$ or $T.width$) which is a sibling of V. This is the situation of an *Inherited Attribute*.



Inherited Attribute



We can handle inherited attributes in one of following ways:

- **[Global]** When we reduce by $T \rightarrow B$, we can remember $T.type$ and $T.width$ in two global variables t and w and use them subsequently
- **[Lazy Action]** Accumulate the list of variables generated from V in a list $V.list$ and the set the type from $T.type$ while reducing with $D \rightarrow T\ V\ \mathbf{;}\ D_1$
- **[Bison Stack]** Use \$0, \$-1 etc. to extract the inherited attribute during reduction of $V \rightarrow$ **id** (or $V \rightarrow V$ **, id**)
- **[Grammar Rewrite]** Rewrite the grammar so that the inherited attributes become synthesized

Attributes for Types: Using Global

type: Type expression for B, T. This is an inherited attribute.

width: The width of a type (B, T), that is, the number of storage units (bytes) needed for objects of that type. It is integral for basic types. This is an inherited attribute.

t: Global to pass the *type* information from a B node to the node for production $V \rightarrow \textbf{id}$.

w: Global to pass the *width* information from a B node to the node for production $V \rightarrow \textbf{id}$.

offset: Global marker for Symbol Table fill-up.

Semantic Actions using Global: Inherited Attributes

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

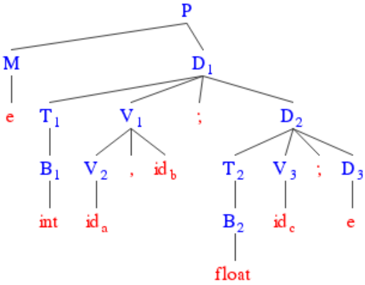
$update(< SymbolTableEntry >, < type >, < width >, < offset >)$ updates the symbol table entry for type, width and offset.

Example: Using Global

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

```
offset = 0
B1.type = integer
B1.width = 4
T1.type = integer
T1.width = 4
t = integer
w = 4
B2.type = float
B2.width = 8
T2.type = float
T2.width = 8
t = float
w = 8
```

Name	Type	Size	Offset
a	integer	4	0
b	integer	4	4
c	float	8	8



Declaration Grammar

0: $P \rightarrow M\ D$

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

2: $D \rightarrow \epsilon$

3: $V \rightarrow V\ ,\ \textbf{id}$

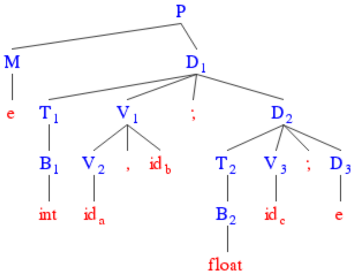
4: $V \rightarrow \textbf{id}$

5: $T \rightarrow B$

6: $B \rightarrow \textbf{int}$

7: $B \rightarrow \textbf{float}$

8: $M \rightarrow \epsilon$



Example: `int a, b; float c;`

Name	Type	Size	Offset
a	int	4	0
b	int	4	4
c	float	8	8

Attributes for Types: Lazy Action

type: Type expression for B, T. This an is inherited (synthesized) attribute.

width: The width of a type (B, T), that is, the number of storage units (bytes) needed for objects of that type. It is integral for basic types. This is an inherited (synthesized) attribute.

list: List of variables generated from V. This is a synthesized attribute.

offset: Global marker for Symbol Table fill-up.

Semantic Actions using Lazy Action: Inherited Attributes

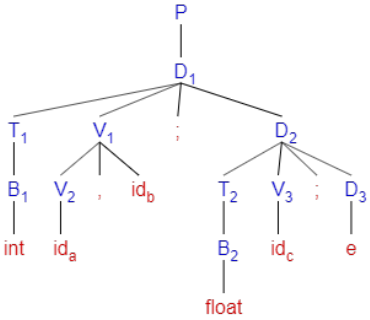
0:	P	$\rightarrow$	D	$\{\ offset = 0;\ update\_offset();\ \}$
1:	D	$\rightarrow$	$T\ \ V\ \ \textbf{;}\ D_1$	$\{\ update(V.list, T.type, T.width);\ \}$
2:	D	$\rightarrow$	ϵ	
3:	V	$\rightarrow$	$V_1\ \textbf{,}\ \textbf{id}$	$\{\ l = makelist(\textbf{id}.loc);$ $V.list = merge(V_1.list, l);\ \}$
4:	V	$\rightarrow$	$\textbf{id}$	$\{\ V.list = makelist(\textbf{id}.loc);\ \}$
5:	T	$\rightarrow$	B	$\{\ T.type = B.type;$ $T.width = B.width;\ \}$
6:	B	$\rightarrow$	$\textbf{int}$	$\{\ B.type = integer; B.width = 4;\ \}$
7:	B	$\rightarrow$	$\textbf{float}$	$\{\ B.type = float; B.width = 8;\ \}$

$update(< ListOfSymbolTableEntry >, < type >, < width >, < offset >)$ updates the symbol table entries on the list for type, width and offset.

$update\_offset();$ updates the offset for all entries in the symbol table

Example: Using Lazy Actions

```
int a, b;    B1.type = integer
float c;     B1.width = 4
             T1.type = integer
             T1.width = 4
             V2.list = {ST[0]}
             V1.list = {ST[0], ST[1]}
             B2.type = float
             B2.width = 8
             T2.type = float
             T2.width = 8
             V3.list = {ST[2]}
             offset = 0
```



States of Symbol Table ST

lists created

	Name	Type	Size	Offset
0	a	?	?	?
1	b	?	?	?
2	c	?	?	?

V3.list resolved

	Name	Type	Size	Offset
0	a	?	?	?
1	b	?	?	?
2	c	float	8	?

V1.list resolved

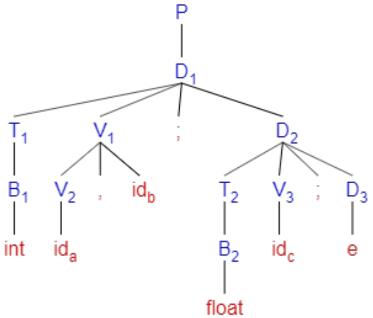
	Name	Type	Size	Offset
0	a	integer	4	?
1	b	integer	4	?
2	c	float	8	?

offsets updated

	Name	Type	Size	Offset
0	a	integer	4	0
1	b	integer	4	4
2	c	float	8	8

Declaration Grammar

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



Example: `int a, b; float c;`

Name	Type	Size	Offset
a	int	4	0
b	int	4	4
c	float	8	8

Attributes for Types: Bison Stack

type: Type expression for B, T. This an inherited attribute.

width: The width of a type (B, T), that is, the number of storage units (bytes) needed for objects of that type. It is integral for basic types. This an inherited attribute.

offset: Global marker for Symbol Table fill-up.

Bison Stack

In the context of:

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

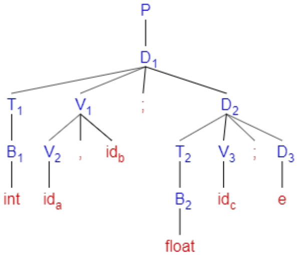
when $V \rightarrow \textbf{id}$ or $V \rightarrow V \ , \ \textbf{id}$ is reduced, the stack is as follows:

id	\$1
T	\$0
…	\$-1
…	\$-2
…	

$V \rightarrow \textbf{id}$

id	\$3
,	\$2
V	\$1
T	\$0
…	\$-1
…	\$-2
…	

$V \rightarrow V \ , \ \textbf{id}$



Semantic Actions using Bison Stack: Inherited Attributes

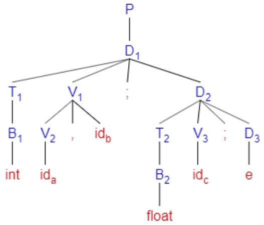
0:	P	$\rightarrow$		$\{\ offset = 0;\ \}$
			D	
1:	D	$\rightarrow$	$T\ \ V\ \mathbf{;}\ D_1$	
2:	D	$\rightarrow$	ϵ	
3:	V	$\rightarrow$	$V\ \mathbf{,}\ \mathbf{id}$	$\{\ update(\mathbf{id}.loc, \$0.type, \$0.width, offset);$ $offset = offset + \$0.width;\ \}$
4:	V	$\rightarrow$	$\mathbf{id}$	$\{\ update(\mathbf{id}.loc,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

$update(< SymbolTableEntry >, < type >, < width >, < offset >)$ updates the symbol table entry for type, width and offset.

Declaration Grammar

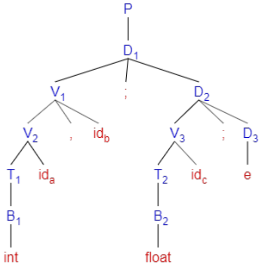
Inherited Attribute

0:	P	$\rightarrow$	D
1:	D	$\rightarrow$	$T\ V\ ;\ D$
2:	D	$\rightarrow$	ϵ
3:	V	$\rightarrow$	V , **id**
4:	V	$\rightarrow$	**id**
5:	T	$\rightarrow$	B
6:	B	$\rightarrow$	**int**
7:	B	$\rightarrow$	**float**



Synthesized Attribute

0:	P	$\rightarrow$	D
1:	D	$\rightarrow$	$V\ ;\ D$
2:	D	$\rightarrow$	ϵ
3:	V	$\rightarrow$	V , **id**
4:	V	$\rightarrow$	T **id**
5:	T	$\rightarrow$	B
6:	B	$\rightarrow$	**int**
7:	B	$\rightarrow$	**float**



Example: `int a, b; float c;`

Name	Type	Size	Offset
a	int	4	0
b	int	4	4
c	float	8	8

Attributes for Types: Grammar Rewrite (Synthesized Attributes)

type: Type expression for B, T, and V. This a synthesized attribute.

width: The width of a type (B, T) or a variable (V), that is, the number of storage units (bytes) needed for objects of that type. It is integral for basic types. This a synthesized attribute.

offset: Global marker for Symbol Table fill-up.

Semantic Actions using Grammar Rewrite: Synthesized Attributes

$$
\begin{array}{llll}
0: & P & \rightarrow & \{\ offset = 0;\ \} \\
 & & & D \\
1: & D & \rightarrow & V\ \mathbf{;}\ D_1 \\
2: & D & \rightarrow & \epsilon \\
3: & V & \rightarrow & V_1\ \mathbf{,}\ \mathbf{id} \\
 & & & \{\ update(\mathbf{id}.loc, V_1.type, V_1.width, offset); \\
 & & & \quad offset = offset + V_1.width; \\
 & & & \quad V.type = V_1.type;\ V.width = V_1.width;\ \} \\
4: & V & \rightarrow & T\ \mathbf{id} \\
 & & & \{\ update(\mathbf{id}.loc, T.type, T.width, offset); \\
 & & & \quad offset = offset + T.width; \\
 & & & \quad V.type = T.type;\ V.width = T.width;\ \} \\
5: & T & \rightarrow & B \\
 & & & \{\ T.type = B.type;\ T.width = B.width;\ \} \\
6: & B & \rightarrow & \mathbf{int}\ \{\ B.type = integer; B.width = 4;\ \} \\
7: & B & \rightarrow & \mathbf{float}\ \{\ B.type = float; B.width = 8;\ \}
\end{array}
$$

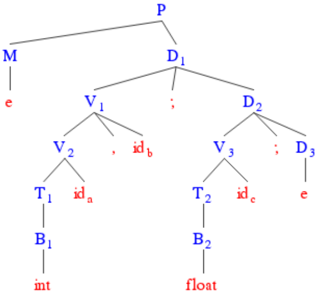
$update(< SymbolTableEntry >, < type >, < width >, < offset >)$ updates the symbol table entry for type, width and offset.

Example: Grammar Rewrite: Synthesized Attributes

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

```
offset = 0
B1.type = integer
B1.width = 4
T1.type = integer
T1.width = 4
V2.type = integer
V2.width = 4
V1.type = integer
V1.width = 4
B2.type = float
B2.width = 8
T2.type = float
T2.width = 8
V3.type = float
V3.width = 8
```

Name	Type	Size	Offset
a	integer	4	0
b	integer	4	4
c	float	8	8



Translation by Type

Use of type in Translation

- **Implicit Conversion**
  - *Safe*
    - Usually smaller type converted to larger type, called *Type Promotion*
    - No data loss
    - Conversions on Type Hierarchy in C:

      ```
      bool -> char -> short int -> int -> unsigned int ->
      long -> unsigned -> long long ->
      float -> double -> long double
      ```
    - Array – Pointer Duality
    - Integer interpreted as Boolean in context
  - *Unsafe*
    - Usually larger type converted to smaller type
    - Potential data loss
- **Explicit Conversion**
  - Using cast operators
  - `void* --> int`, `int --> void*`
- **Type Errors**
  - Between incompatible types

Use of type in Translation: `int` ↔ `double`

Grammar:

$E \rightarrow E_1 + E_2$

$E \rightarrow \textbf{id}$

Translation:

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

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

```
int a, b; double c;
a = b + c; // warning C4244: '=' : conversion from 'double' to 'int',
           // possible loss of data

100: t1 = int2dbl(b)  // Small to Large: Okay
101: t2 = t1 + c
102: t3 = dbl2int(t2) // Large to Small: Data loss
103: a = t3
```

Use of type in Translation: `int` ↔ `double`

```
$E \rightarrow E_1 + E_2$   $\{$ $E.loc = gentemp();$
        $if\,(E_1.type\ != E_2.type)$
            $update(E.loc,$ double, sizeof(double), $offset);$
            $t = gentemp();$
            $update(t,$ double, sizeof(double), $offset);$
            $if\,(E_1.type ==$ integer$)$ // $E_2.type ==$ double
                $emit(t\ '='$ int2dbl$(E_1.loc));$
                $emit(E.loc\ '='\ t\ '+'\ E_2.loc);$
            $else$ // $E_2.type ==$ integer
                $emit(t\ '='$ int2dbl$(E_2.loc));$
                $emit(E.loc\ '='\ E_1.loc\ '+'\ t);$
            $endif$
        $else$
            $update(E.loc, E_1.type,$ sizeof$(E_1.type), offset);$
            $emit(E.loc\ '='\ E_1.loc\ '+'\ E_2.loc);$ $\}$
        $endif$
$E \rightarrow$ **id**   $\{$ $E.loc =$ **id**$.loc;$ $\}$
```

Use of type in Translation: `int` → `bool`

Grammar:

$E \rightarrow E_1$ `!=` E_2

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

$M \rightarrow \epsilon$

$N \rightarrow \epsilon$

Translation:

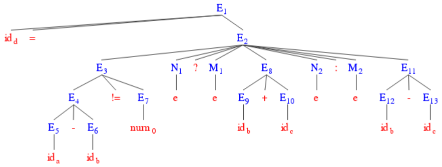
```
int a, b, c, d;
d = a - b != 0 ? b + c : b - c;

100: t1 = a - b
101: t2 = 0
102: if t1 != t2 goto 105
103: goto 107
104: goto 111
105: t3 = b + c
106: goto 110
107: t4 = b - c
108: t5 = t4
109: goto 111
110: t5 = t3
111: d = t5
```

```
int a, b, c, d;
d = a - b ? b + c : b - c;

100: t1 = a - b
101: goto 107
102: t2 = b + c
103: goto 109
104: t3 = b - c
105: t4 = t3
106: goto 110
107: if t1 = 0 goto 104
108: goto 102
109: t4 = t2
110: d = t4
```

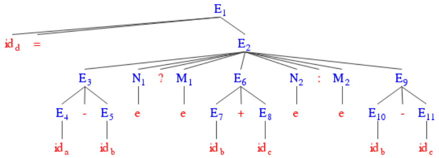
Use of type in Translation: `int` → `bool`

$E \rightarrow E_1$ **!=** $E_2 \mid E_1\ N_1$ **?** $M_1\ E_2\ N_2$ **:** $M_2\ E_3$
$M \rightarrow \epsilon$
$N \rightarrow \epsilon$

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



```
int a, b, c, d; d = a - b ? b + c : b - c;
```



Use of type in Translation: `int` → `bool`

convInt2Bool(E):

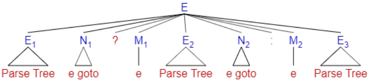
If $E.type$ is integer ($E.loc$ is valid and $E.truelist$ & $E.falselist$ are invalid), it converts $E.type$ to boolean and generates the required codes for it. Now $E.truelist$ and $E.falselist$ become valid and $E.loc$ becomes invalid. Outline of this method is:

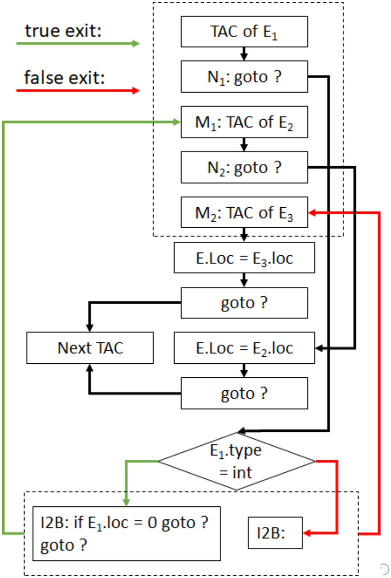
```
$if(E.type ==$ integer$)$
    $E.falselist = makelist(nextinstr);$
    $emit($if $E.loc$ '=' 0 goto .... $);$
    $E.truelist = makelist(nextinstr);$
    $emit($goto .... $);$
$endif$
```

Use of type in Translation: int → bool, bool

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

```
{
  $E.loc = gentemp();$
  // Assume $E_2.type = E_3.type$
  $E.type = E_2.type;$
  // Control gets here by fall-through
  $emit(E.loc$ '=' $E_3.loc);$
  $l = makelist(nextinstr);$
  $emit($goto .... $);$
  $backpatch(N_2.nextlist, nextinstr);$
  $emit(E.loc$ '=' $E_2.loc);$
  $l = merge(l, makelist(nextinstr));$
  $emit($goto .... $);$
  $backpatch(N_1.nextlist, nextinstr);$
  $convInt2Bool(E_1);$
  $backpatch(E_1.truelist, M_1.instr);$
  $backpatch(E_1.falselist, M_2.instr);$
  $backpatch(l, nextinstr);$
}
```





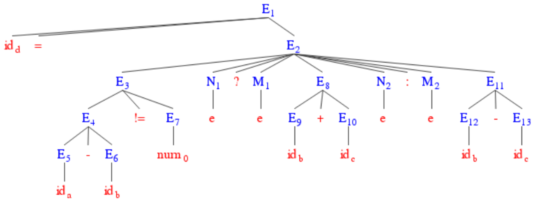
Translation of `?:` for `bool` Condition

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

```
E5.loc = a, E5.type = int
E6.loc = b, E6.type = int
E4.loc = t1, E4.type = int
E7.loc = t2, E7.type = int
E3.type = bool
E3.truelist = {102}
E3.falselist = {103}
N1.nextlist = {104}
M1.instr = 105
E9.loc = b, E9.type = int
E10.loc = c, E10.type = int
E8.loc = t3, E8.type = int
N2.nextlist = {106}
M2.instr = 107
E12.loc = b, E12.type = int
E13.loc = c, E13.type = int
E11.loc = t4, E11.type = int
E2.loc = t5, E2.type = int
E1.loc = t6, E1.type = int
```

```
100: t1 = a - b
101: t2 = 0
102: if t1 != t2 goto 105
103: goto 107
104: goto 112
105: t3 = b + c
106: goto 110
107: t4 = b - c
108: t5 = t4
109: goto 112
110: t5 = t3
111: goto 112
112: d = t5
113: t6 = t5
```

Name	Type	Size	Offset
a	int	4	0
b	int	4	4
c	int	4	8
d	int	4	12
t1	int	4	16
t2	int	4	20
t3	int	4	24
t4	int	4	28
t5	int	4	32
t6	int	4	36



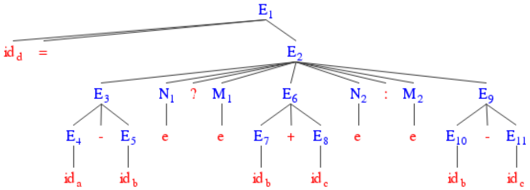
Translation of ?: for int Condition

```
int a, b, c, d; d = a - b ? b + c : b - c;
```

```
E4.loc = a, E4.type = int
E5.loc = b, E5.type = int
E3.loc = t1, E3.type = int
N1.nextlist = {101}
M1.instr = 102
E7.loc = b, E7.type = int
E8.loc = c, E8.type = int
E6.loc = t2, E6.type = int
N2.nextlist = {103}
M2.instr = 104
E10.loc = b, E10.type = int
E11.loc = c, E11.type = int
E9.loc = t3, E9.type = int
E2.loc = t4, E2.type = int
E3.type = bool // Changed
E3.falselist = {109}
E3.truelist = {110}
E1.loc = t5, E1.type = int
```

```
100: t1 = a - b
101: goto 109
102: t2 = b + c
103: goto 107
104: t3 = b - c
105: t4 = t3
106: goto 111
107: t4 = t2
108: goto 111
109: if t1 = 0 goto 104
110: goto 102
111: d = t4
112: t5 = t4
```

Name	Type	Size	Offset
a	int	4	0
b	int	4	4
c	int	4	8
d	int	4	12
t1	int	4	16
t2	int	4	20
t3	int	4	24
t4	int	4	28
t5	int	4	32



Use of type in Translation

for:

```
int i;

for(i = 10; i != 0; --i) { ... } // No conv.

for(i = 10; i; --i) { ... }      // i --> i != 0
```

Grammar / Translation So Far ...

00:	P	$\rightarrow O\ D\ S$
01:	D	$\rightarrow V\ ;\ D$
02:	D	$\rightarrow \epsilon$
03:	V	$\rightarrow V\ ,\ \textbf{id}$
04:	V	$\rightarrow T\ \textbf{id}$
05:	T	$\rightarrow B$
06:	B	$\rightarrow \textbf{int}$
07:	B	$\rightarrow \textbf{float}$
08:	S	$\rightarrow \{\ L\ \}$
09:	S	$\rightarrow \textbf{if}\ (E)\ M\ S_1$
10:	S	$\rightarrow \textbf{if}\ (E)\ M_1\ S_1\ N\ \textbf{else}\ M_2\ S_2$
11:	S	$\rightarrow \textbf{while}\ M_1\ (E)\ M_2\ S_1$
12:	S	$\rightarrow \textbf{do}\ M_1\ S_1\ M_2\ \textbf{while}\ \textbf{(}\ E\ \textbf{);}$
13:	S	$\rightarrow \textbf{for}\ \textbf{(}\ E_1\ \textbf{;}\ M_1\ E\ \textbf{;}\ M_2\ E_2\ N\ \textbf{)}\ M_3\ S_1$
14:	S	$\rightarrow E\ \textbf{;}$
15:	L	$\rightarrow L_1\ M\ S$
16:	L	$\rightarrow S$
17:	E	$\rightarrow E_1\ N_1\ \textbf{?}\ M_1\ E_2\ N_2\ \textbf{:}\ M_2\ E_3$
18:	E	$\rightarrow E_1 = E_2$
19:	E	$\rightarrow E_1\ \vert\vert\ M\ E_2$
20:	E	$\rightarrow E_1\ \&\&\ M\ E_2$
21:	E	$\rightarrow\ !E_1$
22:	E	$\rightarrow E_1\ \texttt{relop}\ E_2$
23:	E	$\rightarrow E_1 + E_2$
24:	E	$\rightarrow E_1 - E_2$
25:	E	$\rightarrow E_1 * E_2$
26:	E	$\rightarrow E_1\ /\ E_2$
27:	E	$\rightarrow (E_1)$
28:	E	$\rightarrow -\ E_1$
29:	E	$\rightarrow \textbf{id}$
30:	E	$\rightarrow \textbf{num}$
31:	E	$\rightarrow \textbf{true}$
32:	E	$\rightarrow \textbf{false}$
33:	O	$\rightarrow \epsilon$
34:	M	$\rightarrow \epsilon$
35:	N	$\rightarrow \epsilon$

Attributes

- E: $E.type$, $E.width$, $E.loc$ ($E.type = \textbf{int}$), $E.truelist$ ($E.type = \textbf{bool}$), $E.falselist$ ($E.type = \textbf{bool}$)
- S: $S.nextlist$
- L: $L.nextlist$
- N: $N.nextlist$
- V: $V.type$, $V.width$
- T: $T.type$, $T.width$
- B: $B.type$, $B.width$
- M: $M.instr$
- **id**: **id**.*loc*
- **num**: **num**.*val*

Handling Arrays in Expression

Arrays in Expression

Translation of Array Expression

array:

```
int a[10], b, i;

b = a[i]; // a[i] --> a + i * sizeof(int)
```

Translation:

```
t1 = i * 4
t2 = a[t1]
b = t2
```

Expression Grammar with Arrays

1: $S \rightarrow \mathbf{id} = E\ ;$

2: $S \rightarrow A = E\ ;$

3: $E \rightarrow E_1 + E_2$

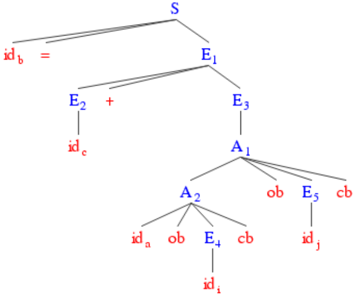
4: $E \rightarrow \mathbf{id}$

5: $E \rightarrow A$

6: $A \rightarrow \mathbf{id}\ [\ E\ ]$

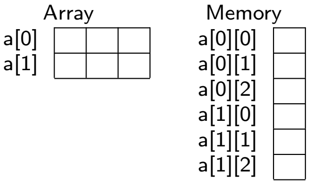
7: $A \rightarrow A_1\ [\ E\ ]$

ob **is** [**and** cb **is**]



Input:

```
int a[2][3], b, c;

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



Output:

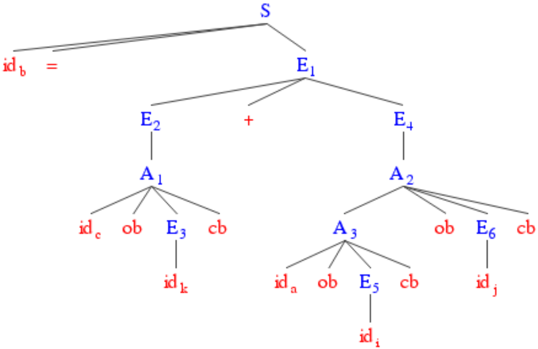
```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a[t3]
t5 = c + t4
b = t5
```

Parse Tree of Array Expression

$$\begin{array}{llll llll}
1: & S & \rightarrow & \textbf{id} = E\ \textbf{;} & 5: & E & \rightarrow & A \\
2: & S & \rightarrow & A = E\ \textbf{;} & 6: & A & \rightarrow & \textbf{id}\ \textbf{[}\ E\ \textbf{]} \\
3: & E & \rightarrow & E_1 + E_2 & 7: & A & \rightarrow & A_1\ \textbf{[}\ E\ \textbf{]} \\
4: & E & \rightarrow & \textbf{id} & & & &
\end{array}$$

`ob` **is** `[` **and** `cb` **is** `]`

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

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



Attributes for Arrays

$A.loc$: Temporary used for computing the offset for the array reference by summing the terms $i_j \times W_j$.

$A.array$: Pointer to the symbol-table entry for the array name. This has *base* and *type*.
The base address of the array, say, $A.array.base$ 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 Arrays

1:	S	$\rightarrow$	$\mathbf{id} = E\ \mathbf{;}$	$\{\ emit(\mathbf{id}.loc\ \text{'='}\ E.loc);\ \}$
2:	S	$\rightarrow$	$A = E\ \mathbf{;}$	$\{\ emit(A.array.base\ \text{'['}\ A.loc\ \text{']'}\ \text{'='}\ E.loc);\ \}$
3:	E	$\rightarrow$	$E_1 + E_2$	$\{\ E.loc = gentemp();\ E.type = E_1.type;$ $emit(E.loc\ \text{'='}\ E_1.loc\ \text{'+'}\ E_2.loc);\ \}$
4:	E	$\rightarrow$	$\mathbf{id}$	$\{\ E.loc = \mathbf{id}.loc;\ E.type = \mathbf{id}.type;\}$
5:	E	$\rightarrow$	A	$\{\ E.loc = gentemp();\ E.type = A.type;$ $emit(E.loc\ \text{'='}\ A.array.base\ \text{'['}\ A.loc\ \text{']'});\ \}$
6:	A	$\rightarrow$	$\mathbf{id}\ \mathbf{[}\ E\ \mathbf{]}$	$\{\ A.array = lookup(\mathbf{id});$ $A.type = A.array.type.elem;$ $A.loc = gentemp();$ $emit(A.loc\ \text{'='}\ E.loc\ \text{'*'}\ A.type.width);\ \}$
7:	A	$\rightarrow$	$A_1\ \mathbf{[}\ E\ \mathbf{]}$	$\{\ A.array = A_1.array;$ $A.type = A_1.type.elem;$ $t = gentemp();$ $A.loc = gentemp();$ $emit(t\ \text{'='}\ E.loc\ \text{'*'}\ A.type.width);$ $emit(A.loc\ \text{'='}\ A_1.loc\ \text{'+'}\ t);\ \}$

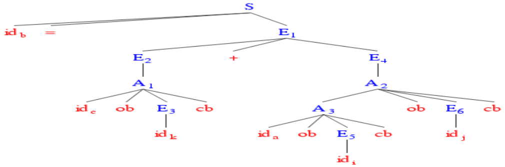
Translation of Array Expression

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

```
E3.loc = k, E3.type = int        .
A1.array = ST[02]                .
A1.type = T2.elem = int          .
A1.loc = t1                      .
A1.loc.type = E3.type = int      100: t1 = k * 4
E2.loc = t2, E2.type = int       101: t2 = c[t1]
E5.loc = i, E5.type = int        .
A3.array = ST[00]                .
A3.type = T1.elem = T2           .
A3.loc = t3                      .
A3.loc.type = E5.type = int      102: t3 = i * 12
E6.loc = j, E6.type = int        .
A2.array = ST[00]                .
A2.type = T2.elem = int          .
A2.loc = t5                      .
A2.loc.type = E6.type = int      103: t4 = j * 4
                                 104: t5 = t3 + t4
E4.loc = t6, E4.type = int       105: t6 = a[t5]
E1.loc = t7, E1.type = int       106: t7 = t2 + t6
                                 107: b = t7
```

No.	Name	Type	Size	Offset
00	a	T1	24	0
01	b	int	4	24
02	c	T2	20	28
03	i	int	4	48
04	j	int	4	52
05	k	int	4	56
06	t1	int	4	16
07	t2	int	4	20
08	t3	int	4	24
09	t4	int	4	28
10	t5	int	4	32
11	t6	int	4	36
12	t7	int	4	36

T1 = array(2, array(3, int)) = array(2, T1')
T2 = array(5, int)
T1 = 2 * T1'.width = 2 * 12 = 24
T1' = 3 * int.width = 3 * 4 = 12



Type Expressions

Declaration Grammar (Inherited Attributes)

0: $P \rightarrow D$
1: $D \rightarrow T\ V\ \mathbf{;}\ D_1$
2: $D \rightarrow \epsilon$
3: $V \rightarrow V_1\ \mathbf{,}\ \mathbf{id}\ \ C$
4: $V \rightarrow \mathbf{id}\ \ C$
5: $T \rightarrow B$
6: $B \rightarrow \mathbf{int}$
7: $B \rightarrow \mathbf{float}$
8: $C \rightarrow \mathbf{[}\ \mathbf{num}\ \mathbf{]}\ C_1$
9: $C \rightarrow \epsilon$

Why the rule of C is right-recursive?
Since the information (of type) needs to flow from the innermost dimension of an array to its outer dimensions (right-to-left), the right recursion is natural. However, while making a reference to that array in an expression, we need to start with its type expression and parse down (left-to-right). Hence, left recursion is natural in $A \rightarrow A\ \mathbf{[}\ E\ \mathbf{]}$.

Symbol Table

Example:

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

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

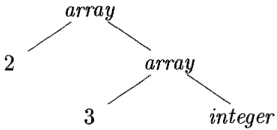
Type Expressions

Applications of types can be grouped under:

- *Type Checking*
  - Logical rules to reason about the behaviour of a program at run time.
  - The types of the operands should match the type expected by an operator. For example, the && operator in Java expects its two operands to be boolean; the result is also of type boolean
- *Translation Applications*
  - Determine the storage that will be needed for that name at run time,
  - Calculate the address denoted by an array reference,
  - Insert explicit type conversions,
  - Choose the right version of an arithmetic operator, ...

Type Expressions

- A *type expression* is either
  - a basic type or
  - formed by applying a *type constructor* operator to a type expression.
- The sets of basic types and constructors depend on the language to be checked.
- *Example*: Type expression of **int[2][3]** (*array of 2 arrays of 3 integers each*) is $array(2, array(3, integer))$



Operator *array* takes two parameters, a *number* and a *type*.

Type Expressions

- *Basic Types*
  - A basic type like **bool**, **char**, **int**, **float**, **double**, or **void** is a type expression. **void** denotes *the absence of a value*.
- *Type Name*
  - A type name is a type expression.
- *Type Constructor*
  - A type expression can be formed by applying the *array* type constructor to a number and a type expression.
  - A **struct** (or record) is a data structure with named fields. A type expression can be formed by applying the *record* type constructor to the field names and their types.
  - For two type expressions s and t, we write type expression $s \rightarrow t$ for *function from type s to type t*, where $\rightarrow$ is a function type constructor.

Type Expressions

- *Cartesian Product*
    - For two type expressions s and t, we write the Cartesian product type expression $s \times t$ to represent a list or tuple of types (like function parameters). $\times$ associates to the left and has precedence over $\rightarrow$.
- *Type Variables*
    - Type expressions may contain variables whose values are type expressions. Compiler-generated type variables are also possible.

struct Type Expression

```
#include <iostream>
using namespace std;

typedef struct {    // record{ name: array (20, char), weight: int}
    char name[20];
    int weight;
} Person;

typedef struct {    // record{ name: array (20, char), weight: int}
    char s_name[20];
    int height;
} Student;

int main() {
    Person p = { "Partha", 80 };
    Student s = { "Arjun", 150 }, t = { "Priyanvada", 120 };

    cout << p.name << " " << p.weight << endl;
    cout << s.s_name << " " << s.height << endl;
    cout << t.s_name << " " << t.height << endl;

    //s = p; // Incompatible types
    s = t; // Compatible types

    cout << s.s_name << " " << s.height << endl;

    return 0;
}
```

Type Equivalence

- *If two type expressions are equal then return a certain type else error.*

```
typedef int * IntPtr;
typedef IntPtr IntPtrArray[10];
typedef int * IPtrArray[10];

IntPtrArray x;
IPtrArray y;
int *z[10];
```

- When type expressions are represented by graphs, two types are structurally equivalent if and only if:
    - They are the same basic type, or
    - They are formed by applying the same constructor to structurally equivalent types, or
    - One is a type name that denotes the other.

Declaration Grammar (Inherited Attributes)

$$
\begin{array}{llll}
0: & P & \rightarrow & D \\
1: & D & \rightarrow & T\ \textbf{id}\ \textbf{;}\ D_1 \\
2: & D & \rightarrow & \epsilon \\
3: & T & \rightarrow & B\ C \\
4: & T & \rightarrow & \textbf{struct}\ \{\ D\ \} \\
5: & B & \rightarrow & \textbf{int} \\
6: & B & \rightarrow & \textbf{float} \\
7: & C & \rightarrow & \textbf{[}\ \textbf{num}\ \textbf{]}\ C_1 \\
8: & C & \rightarrow & \epsilon
\end{array}
$$

For simplicity list of variables in a single declaration has been omitted here.

Attributes for Types

type: – Type expression for B, C.
– This a synthesized attribute.

width: – The width of a type (B, C), that is, the number of storage units (bytes) needed for objects of that type. It is integral for basic types.
– This a synthesized attribute.

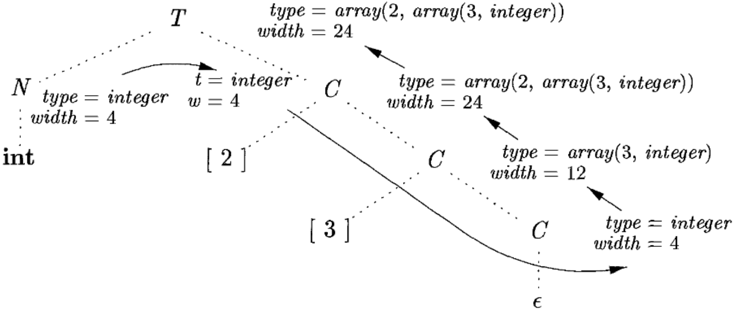
t: – Variable to pass the *type* information from a B node to the node for production $C \rightarrow \epsilon$.
– This an inherited attribute.

w: – Variable to pass the *width* information from a B node to the node for production $C \rightarrow \epsilon$.
– This an inherited attribute.

Computing Types and their Widths

$$\begin{array}{llll}
3: & T \rightarrow & B & \{\ t = B.type;\ w = B.width;\ \} \\
 & & C & \{\ T.type = C.type;\ T.width = C.width;\ \} \\
 & & & \\
5: & B \rightarrow & \textbf{int} & \{\ B.type = integer;\ B.width = 4;\ \} \\
 & & & \\
6: & B \rightarrow & \textbf{float} & \{\ B.type = float;\ B.width = 8;\ \} \\
 & & & \\
7: & C \rightarrow & [\ \textbf{num}\ ]\ C_1 & \\
 & & & \{\ C.type = array(\textbf{num}.value,\ C_1.type); \\
 & & & \quad C.width = \textbf{num}.value \times C_1.width);\ \} \\
 & & & \\
8: & C \rightarrow & \epsilon & \{\ C.type = t;\ C.width = w;\ \}
\end{array}$$

Computing Types and their Widths



Computing Type for int[2][3]

Sequence of Declarations

$$\begin{array}{lllll}
0: & P & \rightarrow & & \{\ \text{offset} = 0;\ \} \\
 & & & D & \\
1: & D & \rightarrow & T\ \textbf{id}\ \textbf{;} & \{\ \mathit{update}(\textbf{id}.\mathit{lexeme}, T.\mathit{type}, \mathit{offset}); \\
 & & & & \quad \mathit{offset} = \mathit{offset} + T.\mathit{width};\ \} \\
 & & & D_1 & \\
2: & D & \rightarrow & \epsilon &
\end{array}$$

Declaration Grammar (Synthesized Attributes)

The translations discussed so far use inherited attributes. We may want to re-write the grammar to use *only* synthesized attributes and in the earlier style design something like:

$$
\begin{array}{llll}
0: & P & \rightarrow & D \\
1: & D & \rightarrow & V \ \mathbf{;} \ D_1 \\
2: & D & \rightarrow & \epsilon \\
3: & V & \rightarrow & V_1 \ \mathbf{,} \ \mathbf{id} \quad C \\
4: & V & \rightarrow & T \ \mathbf{id} \quad C \\
5: & T & \rightarrow & B \\
6: & B & \rightarrow & \mathbf{int} \\
7: & B & \rightarrow & \mathbf{float} \\
8: & C & \rightarrow & \mathbf{[} \ \mathbf{num} \ \mathbf{]} \ C_1 \\
9: & C & \rightarrow & \epsilon
\end{array}
$$

Declaration Grammar (Synthesized Attributes)

- It may be noted that this design is faulty because it still needs inherited attributes to compute the type of C in $C \rightarrow \epsilon$.
- It is rather non-trivial to re-write this grammar for synthesized attributes *only*. This is due to the right-recursive structure of the rules for handling array dimensions. For synthesis, the information naturally flows from left to right while for right recursion the information flows in the reverse order.
- Of course, it is possible to pass this type information through Symbol Table with using explicit global. But that does neither offer an elegant solution.

Functions

Function Declaration Grammar

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

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

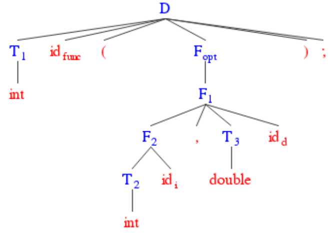
ST(global) *This is the Symbol Table for global symbols*

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: $L \rightarrow L_1 \; S \mid S$

3: $S \rightarrow$ **return** E **;** { Check if function.*type* matches $E.type$;
emit(**return** $E.loc$); }

4: $E \rightarrow$ **id** **(** A_{opt} **)** { $ST = lookup(ST_{gbl}, \textbf{id}).symtab$;
For every param p in $A_{opt}.list$;
Match $p.type$ with param type in ST;
emit(**param** $p.loc$);
$E.loc = gentemp(lookup(ST_{gbl}, \textbf{id}).type)$;
$emit(E.loc = \textbf{call id}, length(A_{opt}.list))$; }

5: $A_{opt} \rightarrow A$ { $A_{opt}.list = A.list$; }

6: $A_{opt} \rightarrow \epsilon$ { $A_{opt}.list = 0$; }

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

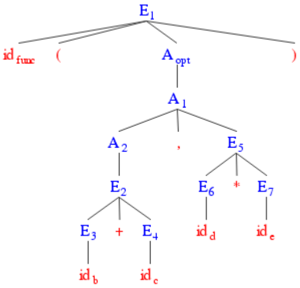
t1	int
t2	double

```
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 = int
E7.loc = e, E7.type = int
E5.loc = t2, E5.type = int
A2.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	→	TU	{ $UpdateOffset(ST_{gbl})$; } // End of TAC Translate
1:	TU	→	$TU_1\ P$	
2:	TU	→	$M\ P$	
3:	M	→	ϵ	{ $ST_{gbl} = CreateSymbolTable()$; $ST_{gbl}.parent = 0$; $cST = ST_{gbl}$; }
4:	P	→	VD	// Variable Declaration
5:	P	→	PD	// Function Prototype Declaration
6:	P	→	FD	// Function Definition
7:	VD	→	$T\ V$ **;**	{ $type_{gbl} = null$; $width_{gbl} = 0$; }
8:	V	→	V_1 **, id** C	{ $Name = lookup(cST, \textbf{id})$; $Name.category = (cST == ST_{gbl})$? global: local; $Name.type = C.type$; $Name.size = C.width$; }
9:	V	→	**id** C	{ $Name = lookup(cST, \textbf{id})$; $Name.category = (cST == ST_{gbl})$? global: local; $Name.type = C.type$; $Name.size = C.width$; }
10:	C	→	**[num]** C_1	{ $C.type = array(\textbf{num}.value, C_1.type)$; $C.width =$**num**$.value \times C_1.width$); }
11:	C	→	ϵ	{ $C.type = type_{gbl}$; $C.width = width_{gbl}$; }
12:	T	→	B	{ $type_{gbl} = T.type = B.type$; $width_{gbl} = T.width = B.width$; }
13:	B	→	**int**	{ $B.type =$ int; $B.width =$ sizeof($B.type$); }
14:	B	→	**double**	{ $B.type =$ double; $B.width =$ sizeof($B.type$); }
15:	B	→	**void**	{ $B.type =$ void; $B.width =$ sizeof($B.type$); }

Grammar for Global, Function and Nested Block Scopes

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

Grammar for Global, Function and Nested Block Scopes

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

Grammar for Global, Function and Nested Block Scopes

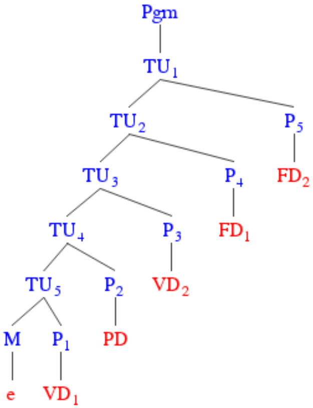
39:	E	$\rightarrow$	**id** **(** AP_{opt} **)**	{ $ST = lookup(ST_{gbl}, \textbf{id}).symtab$; For every param p in $AP_{opt}.list$; Match $p.type$ with param type in ST; $emit(\textbf{param}\ p.loc)$; $E.loc = gentemp(lookup(ST_{gbl}, \textbf{id}).type)$; $emit(E.loc = \textbf{call id,}\ length(AP_{opt}.list))$;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

Example 1: Global & Function Scope: main() & add(): Source

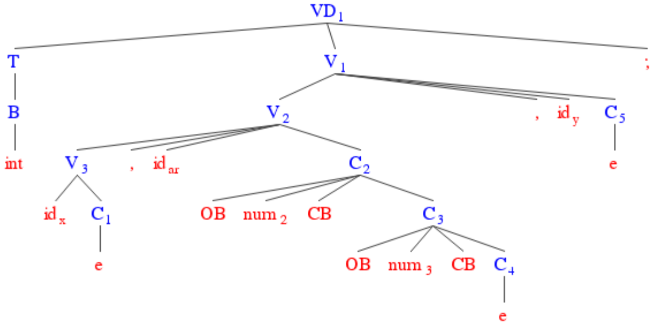
```
int x, ar[2][3], y;
int add(int x, int y);
double a, b;
int add(int x, int y) {
    int t;
    t = x + y;
    return t;
}
void main() {
    int c;
    x = 1;
    y = ar[x][x];
    c = add(x, y);
    return;
}
```

Example 1: Global & Function Scope: Parse Tree (Pgm)

```
                         // M
int x, ar[2][3], y;      // VD_1
int add(int x, int y);   // PD
double a, b;             // VD_2
int add(int x, int y) {  // FD_1
    int t;
    t = x + y;
    return t;
}
void main() {            // FD_2
    int c;
    x = 1;
    y = ar[x][x];
    c = add(x, y);
    return;
}
----
cST = ST.glb
```



Example 1: Global & Function Scope: Parse Tree (VD_1)



```
int x, ar[2][3], y;       // VD_1
```

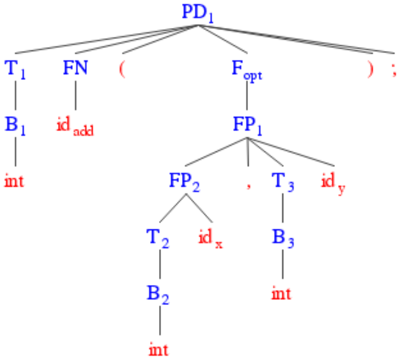
ST.gbl: ST.gbl.parent = null					
x	int	global	4	0	null
ar	array(2, array(3, int))				
		global	24	4	null
y	int	global	4	28	null

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

```
//cST = ST.glb
B.type = int, B.width = 4
T.type = int, T.width = 4
type_glb = int, width_glb = 4
C1.type = int, C1.width = 4
C4.type = int, C4.width = 4
C3.type = array(3, int), C3.width = 12
C2.type = array(2, array(3, int)), C4.width = 24
C5.type = int, C5.width = 4
```

Example 1: Global & Function Scope: Parse Tree (PD_1)

```
//cST = ST.glb
B1.type = int, B1.width = 4
T1.type = int, T1.width = 4
type_glb = int, width_glb = 4
cST = ST.add // FN -> id
B2.type = int, B2.width = 4
T2.type = int, T2.width = 4
type_glb = int, width_glb = 4
B3.type = int, B3.width = 4
T3.type = int, T3.width = 4
type_glb = int, width_glb = 4
cST = ST.glb // PD -> T FN ( F_opt ) ;
```



```
int add(int x, int y);   // PD
```

ST.gbl: ST.gbl.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()

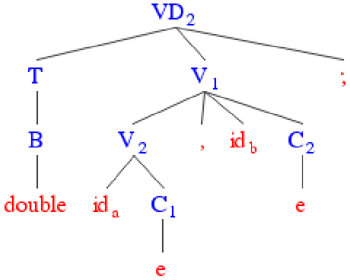
Columns: Name, Type, Category, Size, Offset, & Symtab

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

x	int	param	4	0	
y	int	param	4	4	

Example 1: Global & Function Scope: Parse Tree (VD_2)

```
//cST = ST.glb
B.type = double, B.width = 8
T.type = double, T.width = 8
type_glb = double, width_glb = 8
C1.type = double, C1.width = 8
C2.type = double, C2.width = 8
```



```
double a, b;             // VD_2
```

ST.gbl: ST.gbl.parent = null

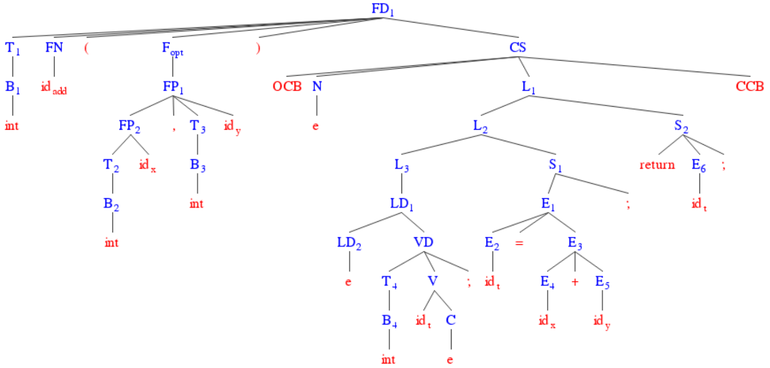
x	int	global	4	0	null
ar	array(2, array(3, int))	global	24	4	null
y	int	global	4	28	null
add	int × int → int	func	0	32	ST.add()
a	double	global	8	32	null
b	double	global	8	40	null

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

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

x	int	param	4	0
y	int	param	4	4

Example 1: Global & Function Scope: Parse Tree (FD_1)



ST.gbl: ST.gbl.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

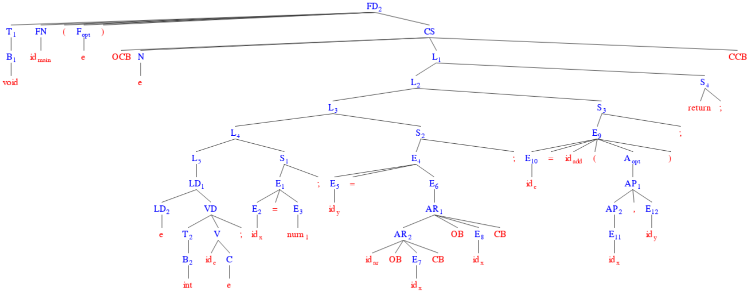
Columns: Name, Type, Category, Size, Offset, &

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

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

```
int add(int x, int y) {  // FD_1
    int t;
    t = x + y;
    return t;
}
```

Example 1: Global & Function Scope: Parse Tree (FD_2)


FD2
T1 FN (Fopt) CS
B1 idmain e OCB N L1 CCB
void e
L2 S4
L3 S3 return ;
L4 S2 E9 ;
L5 S1 E4 ; E10 = idadd (Aopt)
LD1 E1 ; E5 = E6 idc AP1
LD2 VD E2 = E3 idy AR1 AP2 , E12
e T2 V ; idx num1 AR2 OB E8 CB E11 idy
B2 idc C idar OB E7 CB idx idx
int e idx

ST.gbl: ST.gbl.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()

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

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

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

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

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

Example 1: 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.gbl: ST.gbl.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()

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

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

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

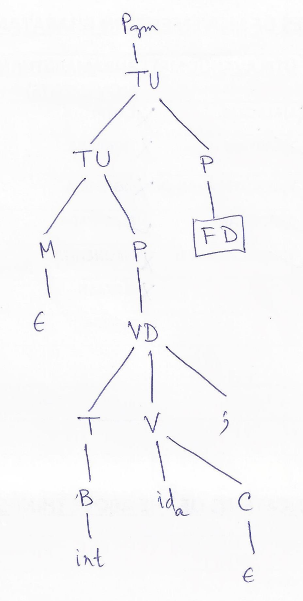
ST.main(): ST.main.parent = ST.gbl

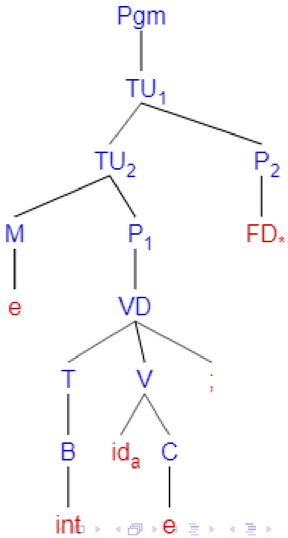
c	int	local	4	0
t#1	int	temp	4	4
t#2	int	temp	4	8
t#3	int	temp	4	12
t#4	int	temp	4	16

Example 2: Nested Blocks: Source

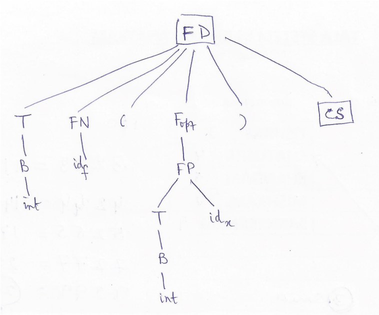
```
int a;
int f(int x) { // function scope f
    int t, u;
    t = x; // t in f, x in f
    { // un-named block scope f_1
         int p, q, t;
         p = a; // p in f_1, a in global
         t = 4; // t in f_1, hides t in f
         { // un-named block scope f_1_1
             int p;
           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

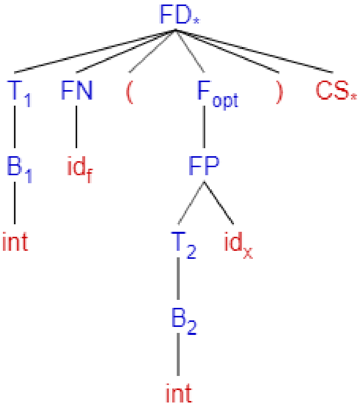
Example 2: Nested Blocks: Parse Tree (Pgm)





Example 2: Nested Blocks: Parse Tree (FD)





FD_*

T_1 FN (F_{opt}) CS_*

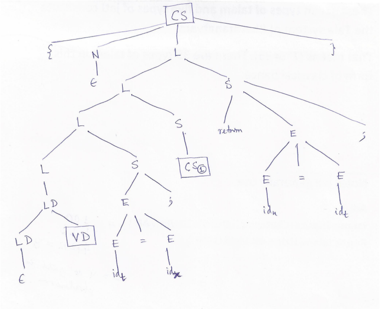
B_1 id_f FP

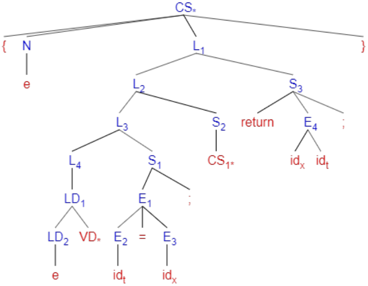
int T_2 id_x

B_2

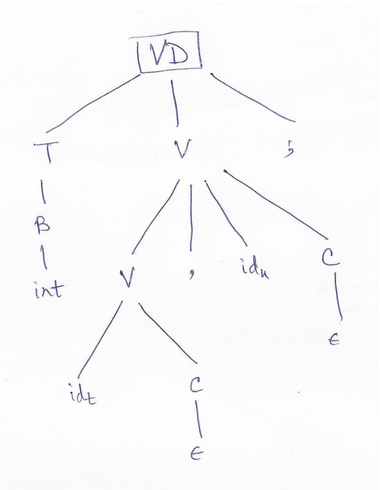
int

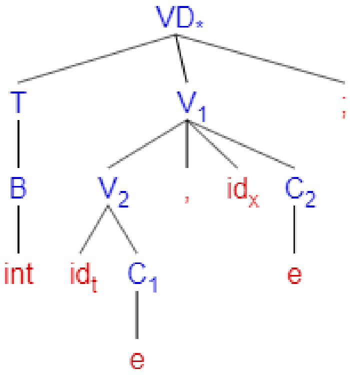
Example 2: Nested Blocks: Parse Tree (CS)



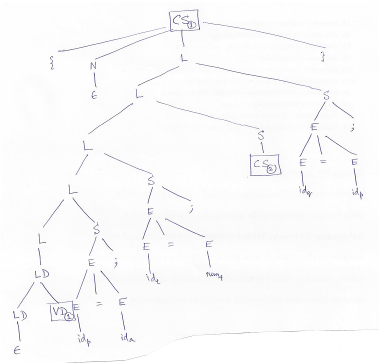


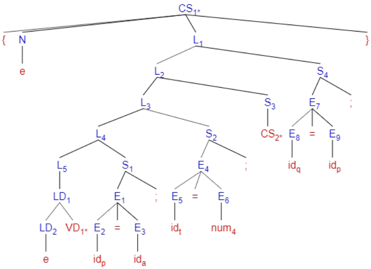
Example 2: Nested Blocks: Parse Tree (VD)



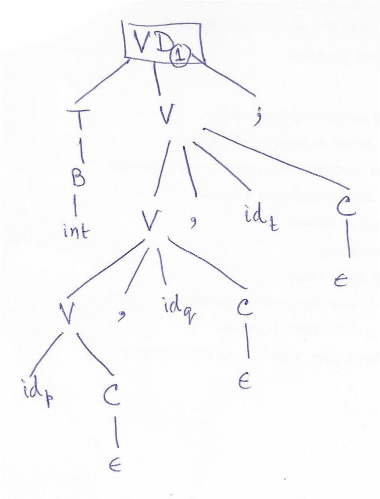


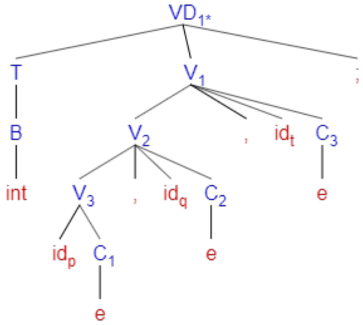
Example 2: Nested Blocks: Parse Tree (CS$_1$)



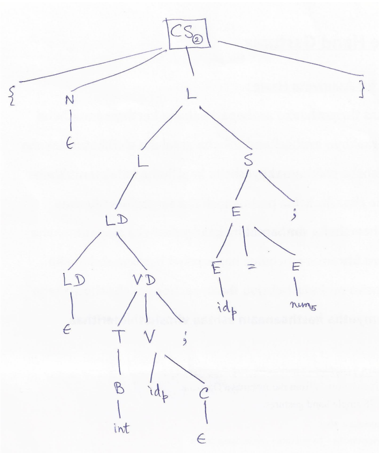


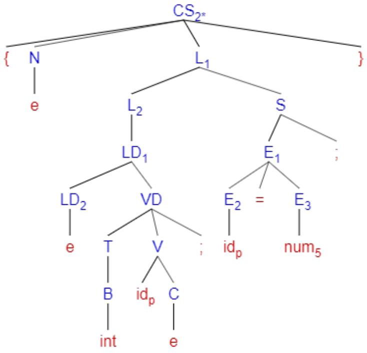
Example 2: Nested Blocks: Parse Tree (VD_1)



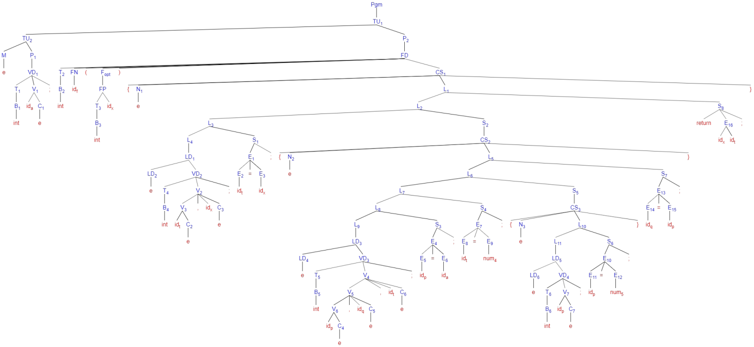


Example 2: Nested Blocks: Parse Tree (CS_2)





Example 2: Nested Blocks: Parse Tree (Pgm Whole)


Pgm
TU_1
TU_2 P_2
M P_1 FD
e VD_1 T_2 FN (F_{opt}) CS_1
T_1 V_1 ; B_2 id_f FP { N_1 L_1 }
B_1 id_a C_1 int T_3 id_x e L_2 S_8
int e B_3 L_3 S_2 return E_{16} ;
int L_4 S_1 CS_2 id_x id_t
LD_1 E_1 ; { N_2 L_5 }
LD_2 VD_2 E_2 = E_3 e L_6 S_7
e T_4 V_2 ; id_t id_x L_7 S_5 E_{13} ;
B_4 V_3 , id_x C_3 L_8 S_4 CS_3 E_{14} = E_{15}
int id_t C_2 e L_9 S_3 E_7 ; { N_3 L_{10} } id_q id_p
e LD_3 E_4 ; E_8 = E_9 e L_{11} S_6
LD_4 VD_3 E_5 = E_6 id_t num_4 LD_5 E_{10} ;
e T_5 V_4 ; id_p id_a LD_6 VD_4 E_{11} = E_{12}
B_5 V_5 , id_t C_6 e T_6 V_7 ; id_p num_5
int V_6 , id_q C_5 e B_6 id_p C_7
id_p C_4 e int e
e

Example 2: 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@gbl
   // 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.gbl: ST.gbl.parent = null

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

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

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

Example 2: 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@gbl
   // 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.f(): ST.f.parent = ST.gbl					
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

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

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

Structures in Expression

Additional Features

Additional Features

- Handling of in C Pre-Processor (CPP)
- Handling of `class` definitions and instantiation
- Handling Inheritance
    - Static
    - Dynamic
- Handling `templates`