Module 05

I Sengupta & P P Das

Module 05: CS31003: Compilers: Machine Independent Translation

Indranil Sengupta
Partha Pratim Das

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

isg@iitkgp.ac.in
ppd@cse.iitkgp.ac.in

September 28 & 29. October 05, 06, 12 & 13, 2020

Module Objectives

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

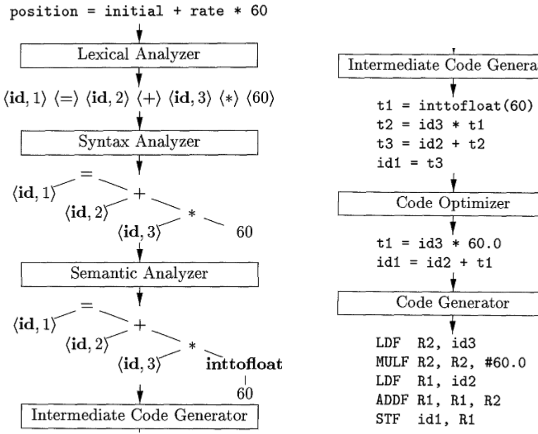
Module Outline

Intermediate Representations

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

Three IRs in Translation



Source: Dragon Book

Figure: Syntax Tree, Three Address Code and Assembly

Alternate Intermediate Representations

- SSA: Single Static Assignment
  - Each variable be assigned exactly once, and
  - Every variable be defined before it is used
- RTL: Register Transfer Language
  - Describes data flow at the register-transfer level of an architecture
- Stack Machines: P-code
- CFG: Control Flow Graph
  - Graph notation
  - All paths in a program during its execution
- DFG: Data Flow Graph
  - Graph notation
  - Data dependancies between a number of operations
- CDFG: Control and Data Flow Graph = CFG + DFG
- 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

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.
  - *Labels*:
    Used to (optionally) mark positions of 3 address instructions

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` ≥ 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.
    Parameters may be stacked in the left-to-right or right-to-left order
  - 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: Notion & Purpose

- Symbol table is a data structure created and maintained by compilers in order to store information about the occurrence of various entities such as variable names, function names, objects, classes, interfaces, etc.
- Symbol table is used by both the analysis and the synthesis parts of a compiler.
- A symbol table may serve the several purposes depending upon the language in hand:
  - To store the names of all entities in a structured form at one place
  - To verify if a variable has been declared
  - To implement type checking, by verifying assignments and expressions in the source code are semantically correct
  - To determine the scope of a name (scope resolution)
- A symbol table is a table which maintains an entry for each name in the following format:

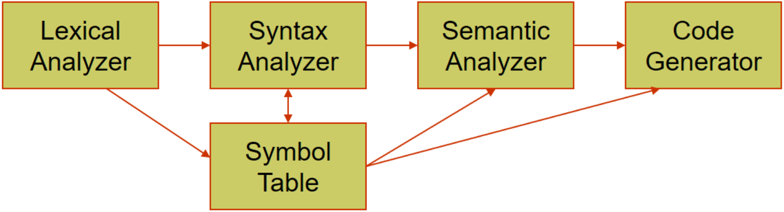
```
<symbol name,  type,  attribute>
```

Symbol Table: Notion & Purpose

- Built in lexical and syntax analysis phases.
- Information collected by the analysis phases of compiler and is used by synthesis phases to generate code.
- Used by compiler to achieve compile time efficiency.
- Used by various phases of compiler as follows:
  - **Lexical Analysis**: Creates new table entries in the table, example like entries about token.
  - **Syntax Analysis**: Adds information regarding attribute type, scope, dimension, line of reference, use, etc.
  - **Semantic Analysis**: Uses information in the table to check for semantics, that is, to verify that expressions and assignments are semantically correct (type checking) and update it accordingly.
  - **Intermediate Code generation**: Refers symbol table for knowing how much and what type of run-time is allocated and table helps in adding temporary variable information.
  - **Code Optimization**: Uses information present in symbol table for machine dependent optimization.
  - **Target Code Generation**: Generates code by using address information of identifier present in the 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.



Note the directionality of arrows with Symbol Table

Symbol Table: Entries

- An ST stores varied information about identifiers:
  - Name (as a string)
    - Name may be qualified for scope or overload resolution
  - Data type (explicit or pointer to Type Table)
  - Block level
  - Scope (global, local, parameter, or temporary)
  - Offset from the base pointer (for local variables and parameters only) – to be used in Stack Frames
  - Initial value (for global and local variables), default value (for parameters)
  - Others (depending on the context)
- A Name (Symbol) may be any one of:
  - Variable (user-define / unnamed temporary)
  - Constant (String and non-String)
  - Function / Method (Global / Class)
  - Alias
  - Type – Class / Structure / Union
  - Namespace

Symbol Table: Scope Rules

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

Static Scoping

```
const int b = 5;

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

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

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

    return 0;
}
```

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

Dynamic Scoping

```
const int b = 5;

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

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

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

    return 0;
}
```

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

Symbol Table: Scope and Visibility

- Scope (visibility) of identifier = portion of program where identifier can be referred to
- Lexical scope = textual region in the program
    - Statement block
    - Method body
    - Class body
    - Module / package / file
    - Whole program (multiple modules)

Symbol Table: Scope and Visibility

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

Symbol Table: Interface

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

Symbol Table: Implementation

- Linear List
- Hash Table
- Binary Search Tree

Example: Global & Function Scopes

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

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

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

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

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

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

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

Example: Global, Function & Block Scopes

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

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

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

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

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

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

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

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

Cols: Name, Type, Category, Size, Offset

- Static Allocation
- Automatic Allocation
- Embedded Automatic Allocation

Example: Global & Function Scopes, typedef


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

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

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

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

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

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

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

Cols: Name, Type, Category, Size, Offset

Example: Global, Function & Class Scopes

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

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

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

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

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

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

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

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

Cols: Name, Type, Category, Size, Offset

More Uses of Symbols Tables

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

Example: Type Symbol Table

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

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

ST.glb				Parent: *Null*
m_dist	class Point × class Point → int			
		func	0	0
p_g	class Point	global	8	
main	int × T_2d_Arr → int			
		func	0	0
ST.m_dist()				Parent: *ST.glb*
q	class Point	param	8	+16
p	class Point	param	8	+8
d	int	local	4	−4
x_diff	int	local	4	−8
y_diff	int	local	4	−12
t1	int	temp	4	−16
t2	int	temp	4	−20
ST.main()				Parent: *ST.glb*
argv	T_2d_Arr	param	4	+8
argc	int	param	4	+4
q	class Point	local	8	−24
dist	int	local	4	−32

ST_type.glb				Parent: *Null*
Point	class Point		8	
Rect	class Rect		16	
T_2d_Arr	arr(*,char*)		4	
ST_type.class Point				Parent: *ST_type.glb*
_x	int	member	4	0
_y	int	member	4	−4
Point	int × int → class Point			
~Point	class Point* → void			
ST_type.class Rect				Parent: *ST_type.glb*
_lt	class Point	member	8	0
_rb	class Point	member	8	−8
Rect	class Point& × class Point& →			
	class Rect	method	0	0
~Rect	class Rect* → void			
get_LT	class Rect* → class Point			
get_RB	class Rect* → class Point			

Cols: Name, Type, Category, Size, Offset

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


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

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

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

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

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

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

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

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

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

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

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

ST.main()				
argv	array(*, char*)	param	4	+8
argc	int	param	4	+4
a	int	local	4	0
b	int	local	4	–4
c	int	local	4	–8

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

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

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

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

ST.glb				
d_add	dbl $\times$ dbl $\rightarrow$ dbl	function	0	0
main	void $\rightarrow$ void	function	0	0

ST.d_add()				
x	dbl	param	8	0
y	dbl	param	8	16
z	dbl	local	8	24

ST.main()				
a	dbl	local	8	0
b	dbl	local	8	8
c	dbl	local	8	16

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

Example: main() & swap()

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
#include <stdio.h>

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

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

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

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

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

ST.glb: ST.glb.parent = null				
Sum	array(*, int) × int → int			
		function	0	ST.Sum
main	void → void	function	0	ST.main
ST.main(): ST.main.parent = ST.glb				
a	array(3, int)	local	12	0
i	int	local	4	12
s	int	local	4	16
n	int	local	4	20
t1	int	temp	4	24

ST.Sum(): ST.Sum.parent = ST.glb				
a	int[]	param	4	0
n	int	param	4	4
i	int	local	4	8
s	int	local	4	12
t_1	int	local	4	16
t1	int	temp	4	20

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

Example: main(), function parameter & other functions

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

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

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

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

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

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

inc:    t1 = x + 1
        return t1

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

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

ST.glb: ST.glb.parent = null				
trans	int × ptr(int → int) × int → int	func	0	0
inc	int → int	func	0	0
dec	int → int	func	0	0
main	void → void	func	0	0
ST.trans(): ST.trans.parent = ST.glb				
a	int	prm	4	0
f	ptr(int → int)	prm	4	4
b	int	prm	4	8
t1	int	tmp	4	12
t2	int	tmp	4	16

ST.inc(): ST.inc.parent = ST.glb				
x	int	prm	4	0
t1	int	tmp	4	4
ST.dec(): ST.dec.parent = ST.glb				
x	int	prm	4	0
t1	int	tmp	4	4
ST.main(): ST.main.parent = ST.glb				
x	int	lcl	4	0
y	int	lcl	4	4
z	int	lcl	4	8
t1	int	tmp	4	12
t2	int	tmp	4	16

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

Example: Nested Blocks: Source & TAC

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

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

ST.glb: ST.glb.parent = null

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

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

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

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

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

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

p	int	local	4	0	null

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

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

Nested Blocks Flattened

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

ST.f(): ST.f.parent = ST.glb					
x	int	param	4	0	null
t	int	local	4	4	null
u	int	local	4	8	null
f_1	null	block	–		ST.f_1

ST.f_1: ST.f_1.parent = ST.f					
p	int	local	4	0	null
q	int	local	4	4	null
t	int	local	4	8	null
f_1_1	null	block	–		ST.f_1_1

ST.f_1_1: ST.f_1_1.parent = ST.f_1					
p	int	local	4	0	null

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

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

ST.f(): ST.f.parent = ST.glb					
x	int	param	4	0	null
t	int	local	4	4	null
u	int	local	4	8	null
p#1	int	blk-local	4	0	null
q#2	int	blk-local	4	4	null
t#3	int	blk-local	4	8	null
p#4	int	blk-local	4	0	null

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


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

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

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

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

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

ST.add(): ST.add.parent = ST.glb				
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.glb				
c	int	local	4	0
t#1	int	temp	4	4
t#2	int	temp	4	8
t#3	int	temp	4	12
t#4	int	temp	4	16

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

Example:Global, Extern & Local Static Data

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

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

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

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

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

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

ST.glb (Main.c)				
n	int	extern	4	0
Sum	int → int	func	0	4
sum	int	global	4	0
main	void → void	func	0	8

ST.glb (Global.c)				
n	int	global	4	0

ST.Sum()				
x	int	param	4	0
lclStcSum	int	static	4	4

ST.main()				
a	int	local	4	0

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

Example: Binary Search

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

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

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

ST.glb				
bs	array(*, int) $\times$ int $\times$ int$\times$ int $\rightarrow$ int	func	0	0

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

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

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

Example: Transpose

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

```
100: t01 = 0
101: i = t01
102: t02 = 3
103: if i < t02 goto 108
104: goto 134
105: t03 = i + 1
106: i = t03
107: goto 103
108: t04 = 0
109: j = t04
110: if j < i goto 115
111: goto 105
112: t05 = j + 1
113: j = t05
114: goto 110
115: t06 = 12 * i
116: t07 = 4 * j
117: t08 = t06 + t07
118: t09 = a[t08]
119: t = t09
120: t10 = 12 * i
121: t11 = 4 * j
122: t12 = t10 + t11
123: t13 = 12 * j
124: t14 = 4 * i
125: t15 = t13 + t14
126: t16 = a[t15]
127: a[t12] = t16
128: t17 = 12 * j
129: t18 = 4 * i
130: t19 = t17 + t18
131: a[t19] = t
132: goto 112
133: goto 105
134: return
```

ST.glb		
main	void → void	func

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

ST.main()				
t08	int	temp	4	–40
t09	int	temp	4	–44
t10	int	temp	4	–48
t11	int	temp	4	–52
t12	int	temp	4	–56
t13	int	temp	4	–60
t14	int	temp	4	–64
t15	int	temp	4	–68
t16	int	temp	4	–72
t17	int	temp	4	–76
t18	int	temp	4	–80
t19	int	temp	4	–84

Arithmetic Expressions

A Calculator Grammar

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

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*, *arg*1, *op*, *arg*2):

– 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

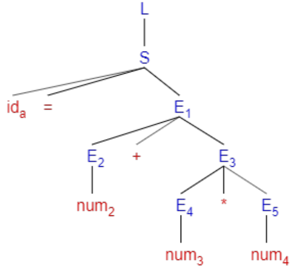
1:	L	$\rightarrow$	$L\ S\ \backslash n$	$\{\ \}$
2:	L	$\rightarrow$	$S\ \backslash n$	$\{\ \}$
3:	S	$\rightarrow$	$\textbf{id} = E$	$\{\ emit(\textbf{id}.loc = E.loc);\ \}$ **// No new temporary, copy code**
4:	E	$\rightarrow$	$E_1 + E_2$	$\{\ E.loc = gentemp();$ $emit(E.loc = E_1.loc + E_2.loc);\ \}$
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**

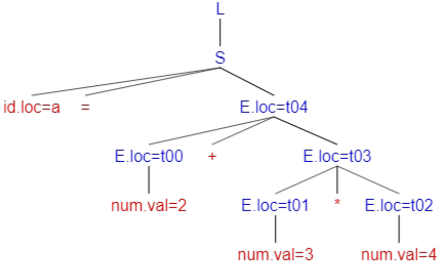
Intermediate 3 address codes are emitted as soon as they are formed.

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





Bison 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
    { $$ = gentemp();
      emit($$->name, $1); }
          ;
%%
```

Bison Section 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

extern YYSTYPE yylval;
```

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

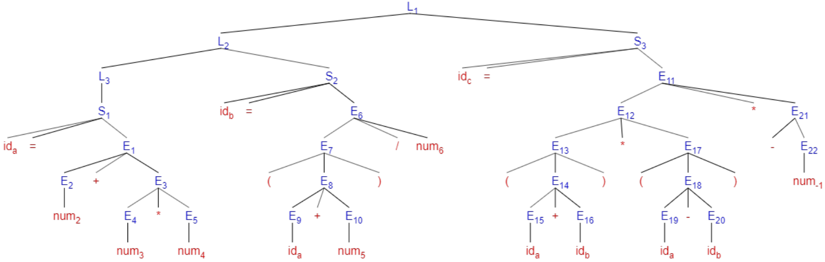

```
%{
#include <math.h>
#include "y.tab.h"
#include "parser.h"
%}

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
$
$
```



Handling of a = 2 + 3 * 4 \n $

Grammar

$$
\begin{array}{lll|lll}
L & \rightarrow & L\ S\ \backslash n & E & \rightarrow & E * E \\
L & \rightarrow & S\ \backslash n & E & \rightarrow & E\ /\ E \\
S & \rightarrow & \textbf{id} = E & E & \rightarrow & (E) \\
E & \rightarrow & E + E & E & \rightarrow & -E \\
E & \rightarrow & E - E & E & \rightarrow & \textbf{num} \\
 & & & E & \rightarrow & \textbf{id}
\end{array}
$$

Reductions

$$
\begin{array}{ll}
 & \textbf{id}_a = \underline{\textbf{num}_2} + \textbf{num}_3 * \textbf{num}_4\ \backslash n \ldots \\
\Rightarrow & \textbf{id}_a = E + \underline{\textbf{num}_3} * \textbf{num}_4\ \backslash n \ldots \\
\Rightarrow & \textbf{id}_a = E + E * \underline{\textbf{num}_4}\ \backslash n \ldots
\end{array}
$$

Stack

num	2	
=		
id	→	"a"

Symtab

a	?

Stack

num	3	
+		
E	→	"t00"
=		
id	→	"a"

Symtab

a	?
t00	?

Stack

E	→	"t01"
+		
E	→	"t00"
=		
id	→	"a"

Symtab

a	?
t00	?
t01	?

Handling of a = 2 + 3 * 4 \n $

Grammar

$$
\begin{array}{lcl|lcl}
L & \rightarrow & L\ S\ \backslash n & E & \rightarrow & E * E \\
L & \rightarrow & S\ \backslash n & E & \rightarrow & E\ /\ E \\
S & \rightarrow & \textbf{id} = E & E & \rightarrow & (E) \\
E & \rightarrow & E + E & E & \rightarrow & -\ E \\
E & \rightarrow & E - E & E & \rightarrow & \textbf{num} \\
 & & & E & \rightarrow & \textbf{id}
\end{array}
$$

Reductions

$$
\begin{array}{ll}
\Rightarrow & \textbf{id}_a = E + E * \underline{\textbf{num}_4}\ \backslash n \ldots \\
\Rightarrow & \textbf{id}_a = E + \underline{E * E}\ \backslash n \ldots \\
\Rightarrow & \textbf{id}_a = \underline{E + E}\ \backslash n \ldots
\end{array}
$$

Stack

num	4	
*		
E	$\rightarrow$	"t01"
+		
E	$\rightarrow$	"t00"
=		
id	$\rightarrow$	"a"

E	$\rightarrow$	"t02"
*		
E	$\rightarrow$	"t01"
+		
E	$\rightarrow$	"t00"
=		
id	$\rightarrow$	"a"

E	$\rightarrow$	"t03"
+		
E	$\rightarrow$	"t00"
=		
id	$\rightarrow$	"a"

Symtab

a	?
t00	?
t01	?

a	?
t00	?
t01	?
t02	?

a	?
t00	?
t01	?
t02	?
t03	?

Handling of a = 2 + 3 * 4 \n $

Grammar

$$
\begin{array}{lcl|lcl}
L & \rightarrow & L\ S\ \backslash\text{n} & E & \rightarrow & E * E \\
L & \rightarrow & S\ \backslash\text{n} & E & \rightarrow & E\ /\ E \\
S & \rightarrow & \textbf{id} = E & E & \rightarrow & (E) \\
E & \rightarrow & E + E & E & \rightarrow & -\ E \\
E & \rightarrow & E - E & E & \rightarrow & \textbf{num} \\
 & & & E & \rightarrow & \textbf{id}
\end{array}
$$

Reductions

$$
\begin{array}{ll}
\Rightarrow & \textbf{id}_a = \underline{E + E}\ \backslash\text{n} \ldots \\
\Rightarrow & \underline{\textbf{id}_a = E}\ \backslash\text{n} \ldots \\
\Rightarrow & \underline{S\ \backslash\text{n}} \ldots \\
\Rightarrow & L \ldots
\end{array}
$$

Stack

E	$\rightarrow$	"t01"
$+$		
E	$\rightarrow$	"t00"
$=$		
id	$\rightarrow$	"a"

E	$\rightarrow$	"t00"
$=$		
id	$\rightarrow$	"a"

\n	
S	

Symtab

a	?
t00	?
t01	?

a	?
t00	?
t01	?
t02	?

a	?
t00	?
t01	?
t02	?
t03	?

Handling of **a = 2 + 3 * 4 \n $**

Grammar

L	$\rightarrow$	$L\ S$ \n	E	$\rightarrow$	$E * E$
L	$\rightarrow$	S \n	E	$\rightarrow$	$E\ /\ E$
S	$\rightarrow$	**id** $= E$	E	$\rightarrow$	(E)
E	$\rightarrow$	$E + E$	E	$\rightarrow$	$-\ E$
E	$\rightarrow$	$E - E$	E	$\rightarrow$	**num**
			E	$\rightarrow$	**id**

Reductions

$\Rightarrow$ $\textbf{id}_a = E + E * \underline{\textbf{num}_4}$ \n ...

$\Rightarrow$ $\textbf{id}_a = E + \underline{E * E}$ \n ...

$\Rightarrow$ $\textbf{id}_a = \underline{E + E}$ \n ...

Stack

num	4	
*		
E	$\rightarrow$	"t01"
+		
E	$\rightarrow$	"t00"
=		
id	$\rightarrow$	"a"

E	$\rightarrow$	"t02"
*		
E	$\rightarrow$	"t01"
+		
E	$\rightarrow$	"t00"
=		
id	$\rightarrow$	"a"

E	$\rightarrow$	"t03"
+		
E	$\rightarrow$	"t00"
=		
id	$\rightarrow$	"a"

Symtab

a	?
t00	?
t01	?

a	?
t00	?
t01	?
t02	?

a	?
t00	?
t01	?
t02	?
t03	?

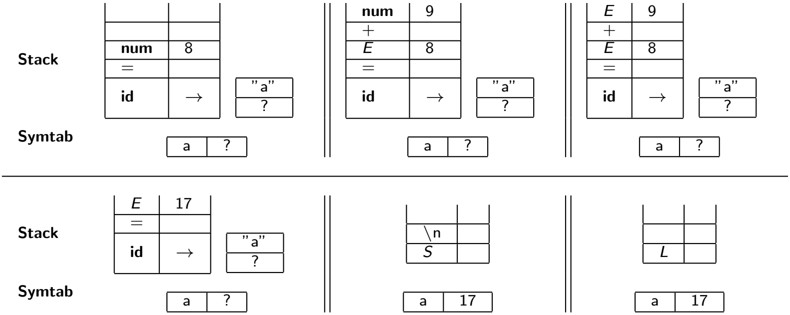
Handling of $\mathbf{a = 8 + 9 \backslash n\ a + 4 \backslash n\ \$}$

Grammar

$$
\begin{array}{lll|lll}
L & \rightarrow & L\ S \ \backslash\text{n} & E & \rightarrow & E * E \\
L & \rightarrow & S \ \backslash\text{n} & E & \rightarrow & E\ /\ E \\
S & \rightarrow & \mathbf{id} = E & E & \rightarrow & (E) \\
S & \rightarrow & E & E & \rightarrow & -\ E \\
E & \rightarrow & E + E & E & \rightarrow & \mathbf{num} \\
E & \rightarrow & E - E & E & \rightarrow & \mathbf{id}
\end{array}
$$

Reductions

$$
\begin{array}{ll}
 & \mathbf{id}_a = \underline{\mathbf{num}_8} + \mathbf{num}_9 \ \backslash\text{n}\ \mathbf{id}_a + \mathbf{num}_4 \ \backslash\text{n}\ \$ \\
\Rightarrow & \mathbf{id}_a = E + \underline{\mathbf{num}_9} \ \backslash\text{n}\ \mathbf{id}_a + \mathbf{num}_4 \ \backslash\text{n}\ \$ \\
\Rightarrow & \mathbf{id}_a = \underline{E + E} \ \backslash\text{n}\ \mathbf{id}_a + \mathbf{num}_4 \ \backslash\text{n}\ \$ \\
\Rightarrow & \underline{\mathbf{id}_a = E} \ \backslash\text{n}\ \mathbf{id}_a + \mathbf{num}_4 \ \backslash\text{n}\ \$ \\
\Rightarrow & \underline{S \ \backslash\text{n}}\ \mathbf{id}_a + \mathbf{num}_4 \ \backslash\text{n}\ \$ \\
\Rightarrow & L\ \underline{\mathbf{id}_a} + \mathbf{num}_4 \ \backslash\text{n}\ \$
\end{array}
$$



Handling of a = 8 + 9 \n a + 4 \n $

Grammar

L	$\rightarrow$	$L\ S$ \n	E	$\rightarrow$	$E * E$
L	$\rightarrow$	S \n	E	$\rightarrow$	$E\ /\ E$
S	$\rightarrow$	**id** $= E$	E	$\rightarrow$	(E)
S	$\rightarrow$	E	E	$\rightarrow$	$-E$
E	$\rightarrow$	$E + E$	E	$\rightarrow$	**num**
E	$\rightarrow$	$E - E$	E	$\rightarrow$	**id**

Reductions

$\Rightarrow L\ \underline{\mathbf{id}_a} + \mathbf{num}_4$ \n $

$\Rightarrow L\ E + \underline{\mathbf{num}_4}$ \n $

$\Rightarrow L\ E + \underline{E}$ \n $

$\Rightarrow L\ \underline{E + E}$ \n $

$\Rightarrow L\ \underline{E}$ \n $

$\Rightarrow \underline{L\ S \text{ \\n}}$ $

$\Rightarrow L$ $

Stack

id	$\rightarrow$
L	

"a"
17

num	4
+	
E	17
L	

E	4
+	
E	17
L	

Symtab

a	17

a	17

a	17

Stack

E	21
L	

\n	
S	
L	

L	

Symtab

a	17

a	17

a	17

Output

= 21

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 Bison 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

Bison 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); }
          ;
%%
```

Bison 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

extern YYSTYPE yylval;
```

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

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

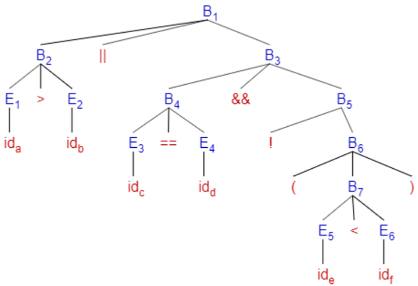
`relop` is any one of:

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

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
```

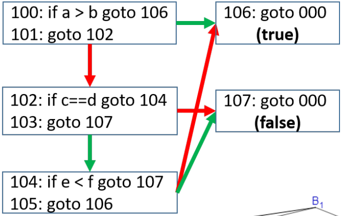


Translation by Value:

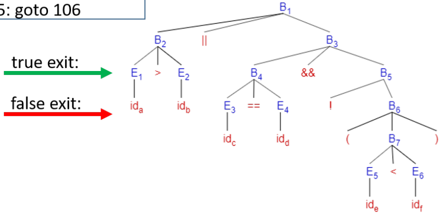
- May not be very useful, as Boolean values are typically used for control flow
- May not use short-cut of computation

Boolean Expression Example: Translation by Control Flow

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




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

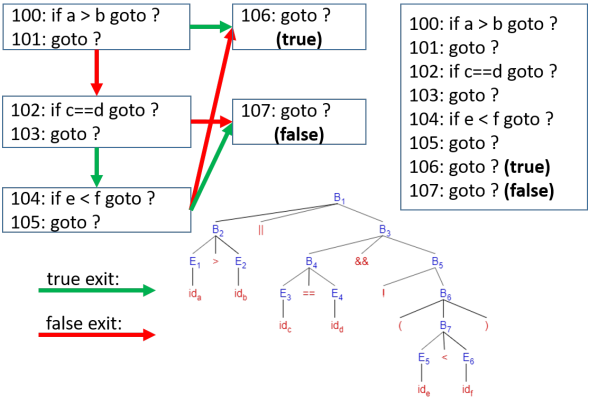


Translation by Control:

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

Boolean Expression Example: Translation by Control Flow

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

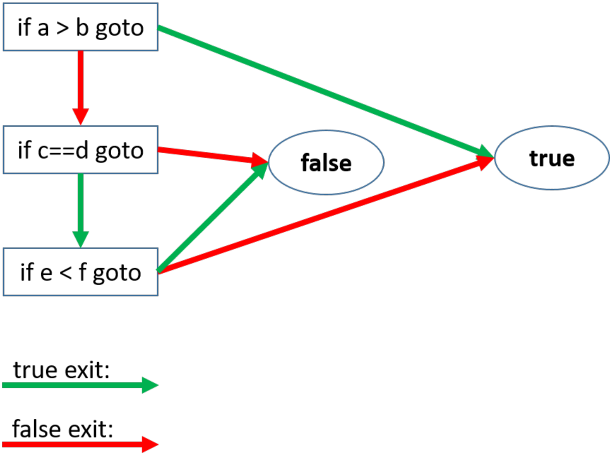


Translation by Control:

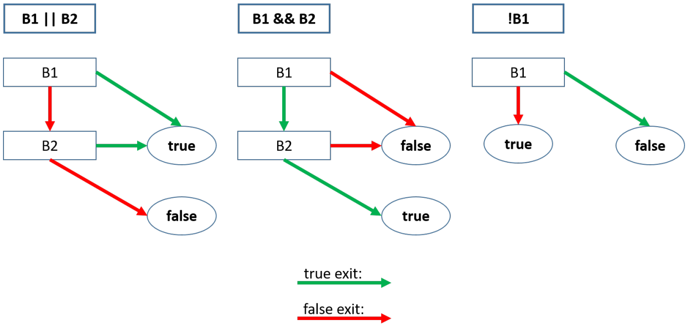
- How to get the target address of goto's?
- Can we optimize goto to goto's / fall-through's

Boolean Expression Example: Translation by Control Flow: Abstracted

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



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 `relop` E_2
6:	B	$\rightarrow$	`true`
7:	B	$\rightarrow$	`false`
8:	M	$\rightarrow$	ϵ // Marker rule

Back-patching Boolean Expression Grammar with Actions

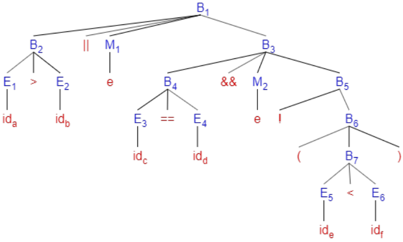
$$
\begin{array}{llll}
1: & B & \rightarrow & B_1 \;||\; M\; B_2 \\
 & & & \quad \{\; backpatch(B_1.falselist, M.instr); \\
 & & & \quad\;\; B.truelist = merge(B_1.truelist, B_2.truelist); \\
 & & & \quad\;\; B.falselist = B_2.falselist; \;\} \\
2: & B & \rightarrow & B_1 \;\&\&\; M\; B_2 \\
 & & & \quad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

Back-patching Boolean Expression Grammar with Actions – Home Assignment

$$
\begin{array}{llll}
9: & B & \rightarrow & B_1 \,\hat{}\, M \; B_2 \\
& & & \quad \{ \; backpatch(B_1.truelist, nextinstr); \\
& & & \quad\;\; emit(B_1.loc, " = ", true); \\
& & & \quad\;\; emit("goto", \; M.instr); \\
& & & \quad\;\; backpatch(B_1.falselist, nextinstr); \\
& & & \quad\;\; emit(B_1.loc, " = ", false); \\
& & & \quad\;\; emit("goto", \; M.instr); \\
& & & \\
& & & \quad\;\; B.truelist = makelist(nextinstr); \\
& & & \quad\;\; backpatch(B_2.falselist, nextinstr); \\
& & & \quad\;\; emit("if", B_1.loc, \; "goto", \; "....."); \\
& & & \quad\;\; B.falselist = makelist(nextinstr); \\
& & & \quad\;\; emit("goto", \; "....."); \\
& & & \\
& & & \quad\;\; temp = makelist(nextinstr); \\
& & & \quad\;\; B.falselist = merge(B.falselist, temp); \\
& & & \quad\;\; backpatch(B_2.truelist, nextinstr); \\
& & & \quad\;\; emit("if", B_1.loc, \; "goto", \; "....."); \\
& & & \quad\;\; temp = makelist(nextinstr); \\
& & & \quad\;\; B.truelist = merge(B.truelist, temp); \\
& & & \quad\;\; emit("goto", \; "....."); \; \}
\end{array}
$$

Example: Boolean Expression

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




```
[1] 100: if a > b goto ?
[1] 101: goto 102            // [9] BP(B2.FL, M1.I)
[3] 102: if c == d goto 104 // [8] 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 #
```

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

Control Constructs

Control Construct Grammar

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

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

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

Back-patching Control Construct Grammar with Actions

1: $S \rightarrow \{ \; L \; \}$ $\quad \{ \; S.nextlist = L.nextlist; \; \}$

2: $S \rightarrow \textbf{id} = \; E \; \textbf{;}$ $\quad \{ \; S.nextlist = null;$
$emit(\textbf{id}.loc, \; '' = '', \; E.loc); \; \}$

3: $S \rightarrow \textbf{if} \; (B) \; M \; S_1$ $\quad \{ \; backpatch(B.truelist, M.instr);$
$S.nextlist = merge(B.falselist, S_1.nextlist); \; \}$

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: $L \rightarrow L_1 \; M \; S$ $\quad \{ \; backpatch(L_1.nextlist, M.instr);$
$L.nextlist = S.nextlist; \; \}$

Back-patching Control Construct Grammar with Actions

7:	L	$\rightarrow$	S	$\{\ L.nextlist = S.nextlist;\ \}$
8:	E	$\rightarrow$	**id**	$\{\ E.loc = \mathbf{id}.loc;\ \}$
9:	E	$\rightarrow$	**num**	$\{\ E.loc = gentemp();$ $emit(E.loc, " = ", num.val);\ \}$
10:	M	$\rightarrow$	ϵ	$\{\ M.instr = nextinstr;\ \}$
11:	N	$\rightarrow$	ϵ	$\{\ N.nextlist = makelist(nextinstr);$ $emit("goto", ".....");\ \}$

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

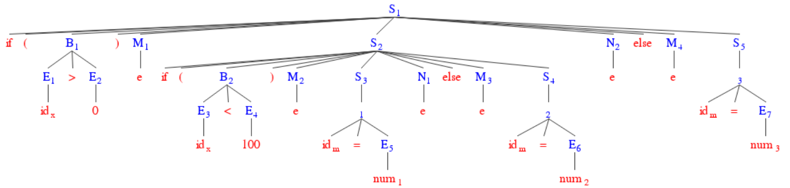
```
if (x > 0) if (x < 100) m = 1; else m = 2; else m = 3;
```

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}
```



[Tutorial] Control Construct Grammar: Extended

$$
\begin{array}{rlll}
1: & S & \rightarrow & \{\ L\ \} \\
2: & S & \rightarrow & \textbf{id} = \ E\ \textbf{;} \\
3: & S & \rightarrow & \textbf{if}\ (B)\ S \\
4: & S & \rightarrow & \textbf{if}\ (B)\ S\ \textbf{else}\ S \\
5: & S & \rightarrow & \textbf{while}\ (B)\ S \\
6: & S & \rightarrow & \textbf{do}\ S\ \textbf{while}\ \textbf{(}\ B\ \textbf{);} \\
7: & S & \rightarrow & \textbf{for}\ \textbf{(}\ E\ \textbf{;}\ B\ \textbf{;}\ E\ \textbf{)}\ S \\
8: & L & \rightarrow & L\ S \\
9: & L & \rightarrow & S \\
10: & E & \rightarrow & \textbf{id} \\
11: & E & \rightarrow & \textbf{num}
\end{array}
$$

[Tutorial] Back-patching Control Construct Grammar: Extended

1: $S \rightarrow \{ \; L \; \}$
2: $S \rightarrow \textbf{id} = \; E \; ;$
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: $S \rightarrow \textbf{do} \; M_1 \; S_1 \; M_2 \; \textbf{while} \; \textbf{(} \; B \; \textbf{);}$
7: $S \rightarrow \textbf{for} \; \textbf{(} \; E_1 \; \textbf{;} \; M_1 \; B \; \textbf{;} \; M_2 \; E_2 \; N \; \textbf{)} \; M_3 \; S_1$
8: $L \rightarrow L_1 \; M \; S$
9: $L \rightarrow S$
10: $E \rightarrow \textbf{id}$
11: $E \rightarrow \textbf{num}$
12: $M \rightarrow \epsilon$ // Marker rule
13: $N \rightarrow \epsilon$ // Fall-through Guard rule

[Tutorial] Back-patching Control Construct Grammar with Actions: Extended

6: $S \rightarrow$ **do** $M_1\ S_1\ M_2$ **while (B);**

$\{$ $backpatch(B.truelist, M_1.instr);$
$backpatch(S_1.nextlist, M_2.instr);$
$S.nextlist = B.falselist;$ $\}$

7: $S \rightarrow$ **for (** E_1 **;** $M_1\ B$ **;** $M_2\ E_2\ N$ **)** $M_3\ S_1$

$\{$ $backpatch(B.truelist, M_3.instr);$
$backpatch(N.nextlist, M_1.instr);$
$backpatch(S_1.nextlist, M_2.instr);$
$emit("goto"\ M_2.instr);$
$S.nextlist = B.falselist;$ $\}$

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);
```

[Tutorial] Back-patching Control Construct Grammar with Actions

$S \rightarrow$ **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:			

[Tutorial] Back-patching Control Construct Grammar with Actions

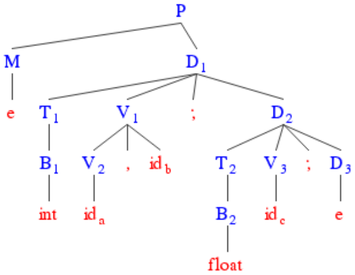
Design suitable schemes to translate **break** and **continue** statements:

$$S \rightarrow \textbf{break;}$$
$$S \rightarrow \textbf{continue;}$$

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

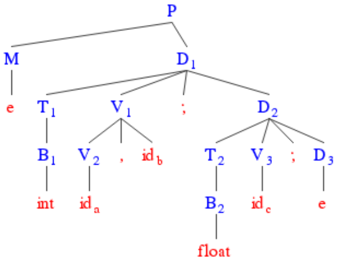
Consider the following attributes for types:

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.

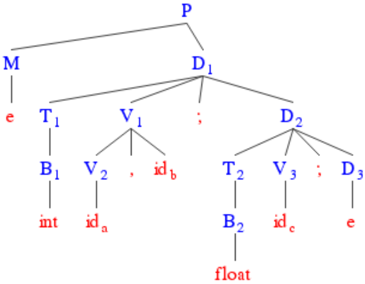
In the context of:

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

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

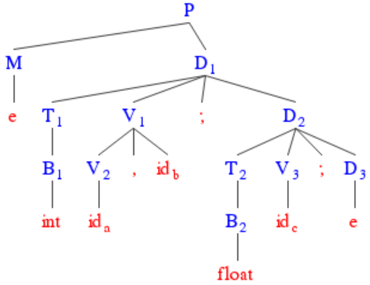
$update(< SymbolTableEntry >, < type >, < width >, < offset >)$ updates the symbol table entry for type, width and offset.

Example: Using Global

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

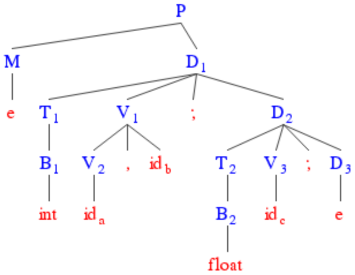
```
offset = 0
B1.type = integer
B1.width = 4
T1.type = integer
T1.width = 4
t = integer
w = 4
B2.type = float
B2.width = 8
T2.type = float
T2.width = 8
t = float
w = 8
```

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	$\rightarrow$	ϵ
3:	V	$\rightarrow$	V **, id**
4:	V	$\rightarrow$	**id**
5:	T	$\rightarrow$	B
6:	B	$\rightarrow$	**int**
7:	B	$\rightarrow$	**float**
8:	M	$\rightarrow$	ϵ



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

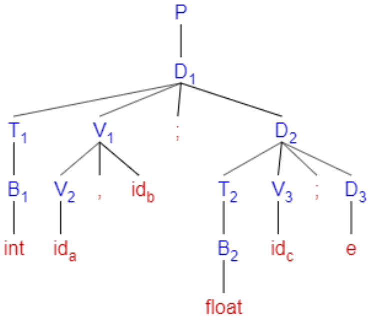
0:	P	$\rightarrow$	D	$\{ \ offset = 0; \ update\_offset(); \ \}$
1:	D	$\rightarrow$	$T \ V \ ; \ D_1$	$\{ \ update(V.list, T.type, T.width); \ \}$
2:	D	$\rightarrow$	ϵ	
3:	V	$\rightarrow$	$V_1 \ , \ \mathbf{id}$	$\{ \ l = makelist(\mathbf{id}.loc);$ $V.list = merge(V_1.list, l); \ \}$
4:	V	$\rightarrow$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

$update(< ListOfSymbolTableEntry >, < type >, < width >, < offset >)$ 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

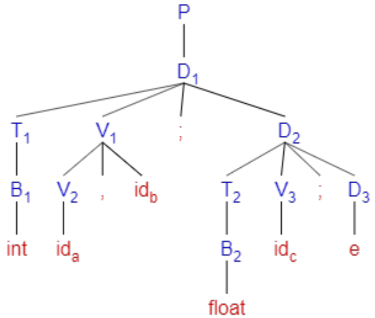
	Name	Type	Size	Offset
0	a	integer	4	?
1	b	integer	4	?
2	c	float	8	?

offsets updated

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

Declaration Grammar

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



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.

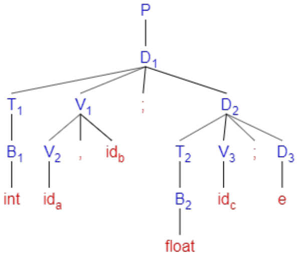
offset: Global marker for Symbol Table fill-up.

Bison Stack

In the context of:

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

when $V \rightarrow$ **id** or $V \rightarrow V$ **, id** is reduced, the stack is as follows:



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

$V \rightarrow$ **id** $\qquad V \rightarrow V$ **, id**

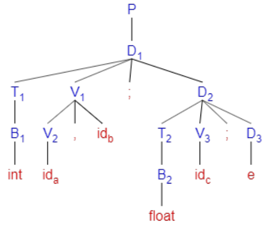
Semantic Actions using Bison Stack: Inherited Attributes

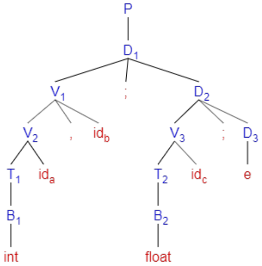
0:	P	$\rightarrow$		$\{\ \mathit{offset} = 0;\ \}$
			D	
1:	D	$\rightarrow$	$T\ V\ \mathbf{;}\ D_1$	
2:	D	$\rightarrow$	ϵ	
3:	V	$\rightarrow$	$V\ \mathbf{,}\ \mathbf{id}$	$\{\ \mathit{update}(\mathbf{id}.\mathit{loc}, \$0.\mathit{type}, \$0.\mathit{width}, \mathit{offset});$ $\mathit{offset} = \mathit{offset}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

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

Declaration Grammar

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





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

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

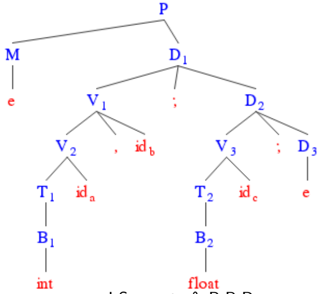
$\mathit{update}(< \mathit{SymbolTableEntry} >, < \mathit{type} >, < \mathit{width} >, < \mathit{offset} >)$ updates the symbol table entry for type, width and offset.

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` $\leftrightarrow$ `double`

Grammar:

$E \rightarrow E_1 + E_2$

$E \rightarrow \mathbf{id}$

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` $\leftrightarrow$ `double`

$$
\begin{array}{llll}
E & \rightarrow & E_1 + E_2 & \{\ E.loc = gentemp(); \\
 & & & \quad if\,(E_1.type\ != E_2.type) \\
 & & & \qquad update(E.loc, \text{double}, \text{sizeof(double)}, offset); \\
 & & & \qquad t = gentemp(); \\
 & & & \qquad update(t, \text{double}, \text{sizeof(double)}, offset); \\
 & & & \qquad if\,(E_1.type == \text{integer})\ //\ E_2.type == \text{double} \\
 & & & \qquad\quad emit(t\ \text{'='}\ \text{int2dbl}(E_1.loc)); \\
 & & & \qquad\quad emit(E.loc\ \text{'='}\ t\ \text{'+'}\ E_2.loc); \\
 & & & \qquad else\ //\ E_2.type == \text{integer} \\
 & & & \qquad\quad emit(t\ \text{'='}\ \text{int2dbl}(E_2.loc)); \\
 & & & \qquad\quad emit(E.loc\ \text{'='}\ E_1.loc\ \text{'+'}\ t); \\
 & & & \qquad endif \\
 & & & \quad else \\
 & & & \qquad update(E.loc, E_1.type, \text{sizeof}(E_1.type), offset); \\
 & & & \qquad emit(E.loc\ \text{'='}\ E_1.loc\ \text{'+'}\ E_2.loc);\ \} \\
 & & & \quad endif \\
E & \rightarrow & \mathbf{id} & \{\ E.loc = \mathbf{id}.loc;\ \}
\end{array}
$$

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

Grammar:

$E \rightarrow E_1 \texttt{ != } E_2$

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

$M \rightarrow \epsilon$

$N \rightarrow \epsilon$

Translation:

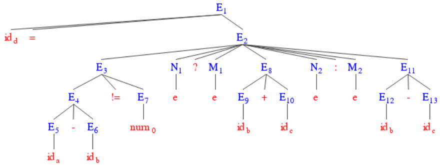
```
int a, b, c, d;
d = a - b != 0 ? b + c : b - c;

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
```

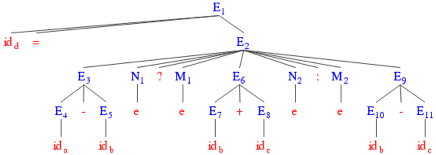
Use of type in Translation: `int` $\rightarrow$ `bool`

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

```
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: $\texttt{int} \rightarrow \texttt{bool}$

$convInt2Bool(E)$:

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` $\rightarrow$ `bool`

$$E \quad \rightarrow \quad E_1 \; N_1 \; \textbf{?} \; M_1 \; E_2 \; N_2 \; \textbf{:} \; M_2 \; E_3$$

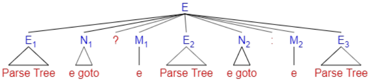
$\{$ $E.loc = gentemp()$;
$E.type = E_2.type$; // Assume $E_2.type = E_3.type$
$emit(E.loc \text{ '=' } E_3.loc)$; // Control gets here by fall-through
$l = makelist(nextinstr)$;
$emit(\text{goto .... })$;
$backpatch(N_2.nextlist, nextinstr)$;
$emit(E.loc \text{ '=' } E_2.loc)$;
$l = merge(l, makelist(nextinstr))$;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

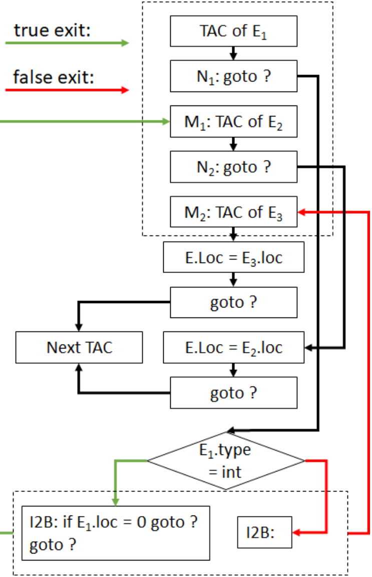
Use of type in Translation: `int` $\rightarrow$ `bool`, `bool`

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

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

}





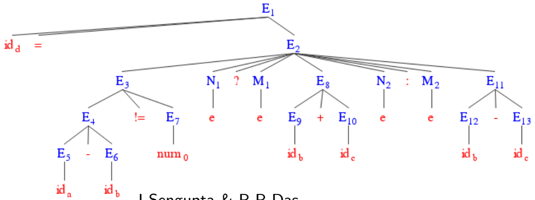
Translation of `?:` for `bool` Condition

```
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



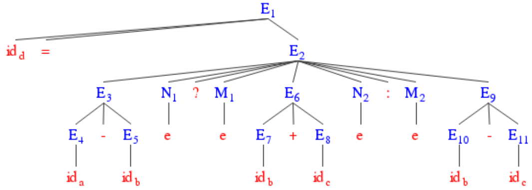
Translation of ?: for int Condition

```
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	$\rightarrow E_1\ \|\|\ M\ E_2$
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

Arrays in Expression

Translation of Array Expression

array:

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

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

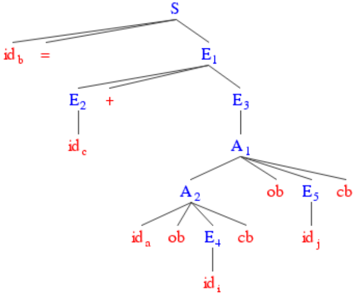
Translation:

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

Expression Grammar with Arrays

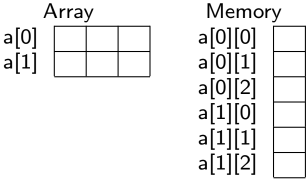
1: $S \rightarrow \textbf{id} = E\ \textbf{;}$
2: $S \rightarrow A = E\ \textbf{;}$
3: $E \rightarrow E_1 + E_2$
4: $E \rightarrow \textbf{id}$
5: $E \rightarrow A$
6: $A \rightarrow \textbf{id}\ \textbf{[}\ E\ \textbf{]}$
7: $A \rightarrow A_1\ \textbf{[}\ E\ \textbf{]}$

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



Input:

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

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



Output:

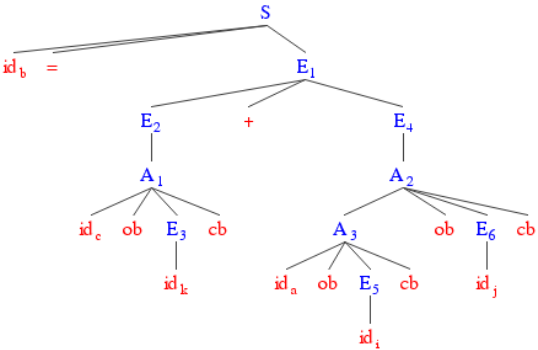
```
t1 = i * 12
t2 = j * 4
t3 = t1 + t2
t4 = a[t3]
t5 = c + t4
b = t5
```

Parse Tree of Array Expression

1:	$S \rightarrow$	**id** $=$ E **;**		5:	$E \rightarrow$	A
2:	$S \rightarrow$	$A = E$ **;**		6:	$A \rightarrow$	**id [** E **]**
3:	$E \rightarrow$	$E_1 + E_2$		7:	$A \rightarrow$	A_1 **[** E **]**
4:	$E \rightarrow$	**id**				

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

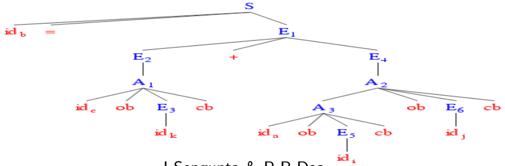
Translation of Array Expression

```
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 = T1'                .
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 = T1'.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')
T1' = array(3, int). T2 = array(5, int)
T1'.width = 3 * int.width = 3 * 4 = 12
T1.width = 2 * T1'.width = 2 * 12 = 24
T2.width = 5 * int.width = 5 * 4 = 20



Expression Grammar with Arrays

Input:

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

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

Output:

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

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

Type Expressions

Declaration Grammar (Inherited Attributes)

Without Array

0:	P	$\rightarrow$	D
1:	D	$\rightarrow$	$T\ V\ \mathbf{;}\ D_1$
2:	D	$\rightarrow$	ϵ
3:	V	$\rightarrow$	$V_1\ \mathbf{,}\ \mathbf{id}$
4:	V	$\rightarrow$	$\mathbf{id}$
5:	T	$\rightarrow$	B
6:	B	$\rightarrow$	**int**
7:	B	$\rightarrow$	**float**

With Array

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

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 in $C \rightarrow \mathbf{[}\ \mathbf{num}\ \mathbf{]}\ C$.

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;
double 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, double)	40	56

sizeof(int) $= 4$
sizeof(double) $= 8$

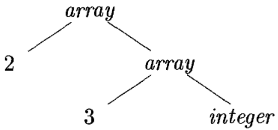
Type Expressions

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.
- *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.

Type Expressions

- *Type Constructor*
  - A type expression can be formed by applying the *array* type constructor to a number and a type expression.

    ```
    int a[10][5];
    ```

    Type ≡ array(10, array (5, int))
  - 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.

    ```
    struct _ {
        char name[20];
        int height;
    }
    ```

    Type ≡ record{name: char[20], height: int}

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 Expressions

- *Type Constructor*
  - 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.

    `int f(int);`

    Type $\equiv$ int $\rightarrow$ int

    `int add(int, int);`

    Type $\equiv$ int $\times$ int $\rightarrow$ int

    `int main(int argc, char *argv[]);`

    Type $\equiv$ int $\times$ array(*, char*) $\rightarrow$ int
  - For a type expression t, address(t) is the expression for its pointer / address type

    `int *p;`

    Type $\equiv$ address(int)

Type Equivalence

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

```
typedef int * IntPtr;           // IntPtr = address(int)
typedef IntPtr IntPtrArray[10]; // IntPtrArray = array(10, IntPtr)
                                //     = array(10, address(int))
typedef int * IPtrArray[10];    // IPtrArray = array(10, address(int))

IntPtrArray x; // IntPtrArray
IPtrArray y;   // IPtrArray
int *z[10];    // T = array(10, address(int))

So, IntPtrArray = IPtrArray = T = array(10, address(int))
```

Further,

```
typedef int (*fptr)(int, int);
int f(int, int);

fptr = address(int X int --> int)
T1 = Type(&f) = address(int X int --> int)}
T2 = Type(f) = int X int --> int

So, fptr = T1, and T2 considered equivalent as well
```

- 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)

With Array

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

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

Attributes for Types

type: – Type expression for B, C, and T.
– This a synthesized attribute for B & C, and an inherited attribute for T.

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

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

w: – Global variable to pass the *width* information from a B node to the node for production $C \rightarrow \epsilon$. $w = t.width$
– This is for handling inherited attribute.

Sequence of Declarations

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

Computing Types and their Widths

$$5: \quad T \rightarrow B \qquad \{\ t = B.type;\ w = B.width;\ \}$$

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

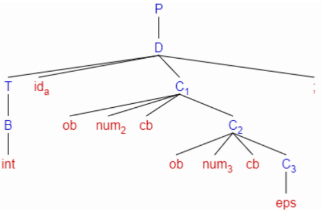
$$7: \quad B \rightarrow \textbf{float} \qquad \{\ B.type = float;\ B.width = 8;\ \}$$

$$8: \quad C \rightarrow \textbf{[num]}\ C_1 \qquad \{\ C.type = array(\textbf{num}.value, C_1.type);\ C.width = \textbf{num}.value \times C_1.width);\ \}$$

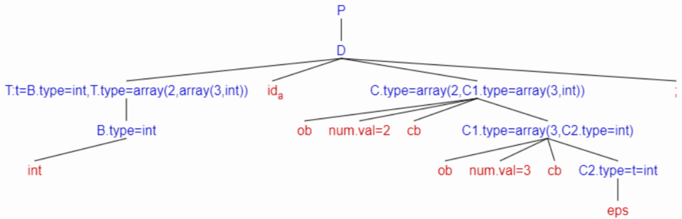
$$9: \quad C \rightarrow \epsilon \qquad \{\ C.type = t;\ C.width = w;\ \}$$

Array Declaration Example

Consider the array declaration: `int a[2][3];`



The parse tree is annotated with the attributes computed by the semantic actions. Note how `t` is set in node T and is used in node $C2$.



Declaration Grammar (Inherited Attributes): Dragon Book

The following grammar and actions are taken from Dragon Book which presents a little different treatment

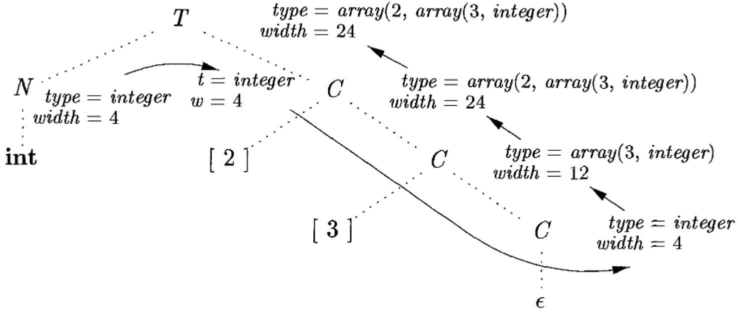
$$
\begin{array}{llll}
0: & P & \rightarrow & D \\
1: & D & \rightarrow & T \ \textbf{id} \ \textbf{;} \ D_1 \\
2: & D & \rightarrow & \epsilon \\
3: & T & \rightarrow & B \ \ C \\
4: & T & \rightarrow & \textbf{struct} \ \{ \ D \ \} \\
5: & B & \rightarrow & \textbf{int} \\
6: & B & \rightarrow & \textbf{float}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

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

Computing Types and their Widths: Dragon Book

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

Computing Types and their Widths: Dragon Book



Computing Type for int[2][3]

The above diagram is taken from the Dragon Book.

Please read the non-terminal N as non-terminal B in our grammar.

Sequence of Declarations: Dragon Book

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

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:

Inherited Attributes				**Synthesized Attributes**			
0:	P	$\rightarrow$	D	0:	P	$\rightarrow$	D
1:	D	$\rightarrow$	$T\ V$ **;** D_1	1:	D	$\rightarrow$	V **;** D_1
2:	D	$\rightarrow$	ϵ	2:	D	$\rightarrow$	ϵ
3:	V	$\rightarrow$	V_1 **, id** C	3:	V	$\rightarrow$	V_1 **, id** C
4:	V	$\rightarrow$	**id** C	4:	V	$\rightarrow$	T **id** C
5:	T	$\rightarrow$	B	5:	T	$\rightarrow$	B
6:	B	$\rightarrow$	**int**	6:	B	$\rightarrow$	**int**
7:	B	$\rightarrow$	**float**	7:	B	$\rightarrow$	**float**
8:	C	$\rightarrow$	**[num]** C_1	8:	C	$\rightarrow$	**[num]** C_1
9:	C	$\rightarrow$	ϵ	9:	C	$\rightarrow$	ϵ

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

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

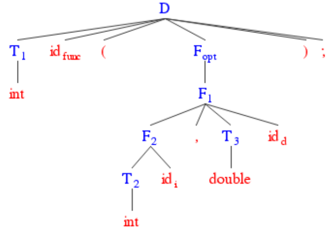
ST(global) *This is the Symbol Table for global symbols*

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

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

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

Function Declaration Example




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

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

ST(global)

Name	Type	Size	Offset	Nested Table
func	int $\times$ dbl $\rightarrow$ 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}	$\rightarrow$	ϵ	{ $A_{opt}.list = 0$; }
7:	A	$\rightarrow$	A_1 **,** E	{ $A.list = Merge(A_1.list, Makelist(E.loc, E.type))$; }
8:	A	$\rightarrow$	E	{ $A.list = Makelist(E.loc, E.type)$; }

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

List of Params

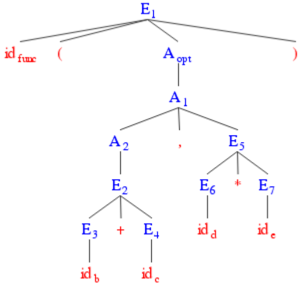
t1	int
t2	double

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

Function Invocation Example

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

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

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

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



ST(global)

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

ST(func)

Name	Type	Size	Offset	Nested Table
i	**int**	4	0	null
d	**dbl**	8	4	null
__*rv*	**int**	4	12	null

ST(?)

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

Lexical Scope Management

Grammar for Global, Function and Nested Block Scopes

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

Grammar for Global, Function and Nested Block Scopes

16:	PD	$\rightarrow$	$T\ FN\ \mathbf{(}\ FP_{opt}\ \mathbf{)};$	$\{$ $UpdateOffset(cST)$; $cST = cST.parent$; $\}$
17:	FD	$\rightarrow$	$T\ FN\ \mathbf{(}\ FP_{opt}\ \mathbf{)}\ CS$	$\{$ $UpdateOffset(cST)$; $cST = cST.parent$; $\}$

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

Grammar for Global, Function and Nested Block Scopes

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

Grammar for Global, Function and Nested Block Scopes

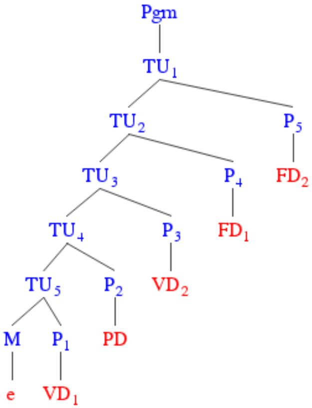
39:	E	$\rightarrow$	**id** **(** AP_{opt} **)**	{ $ST = lookup(ST_{gbl}, \textbf{id}).symtab$; For every param p in $AP_{opt}.list$; Match $p.type$ with param type in ST; $emit(\textbf{param}\ p.loc)$; $E.loc = gentemp(lookup(ST_{gbl}, \textbf{id}).type)$; $emit(E.loc = \textbf{call id},\ length(AP_{opt}.list))$;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

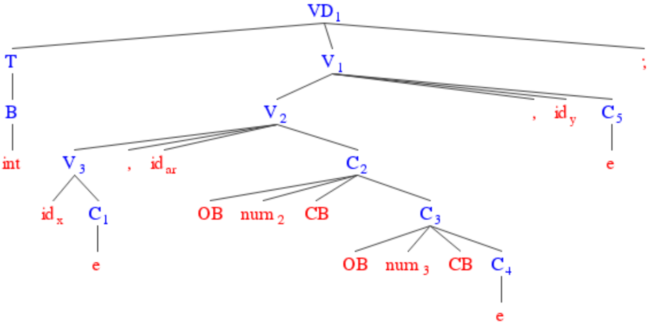

```
int x, ar[2][3], y;
int add(int x, int y);
double a, b;
int add(int x, int y) {
    int t;
    t = x + y;
    return t;
}
void main() {
    int c;
    x = 1;
    y = ar[x][x];
    c = add(x, y);
    return;
}
```

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
```

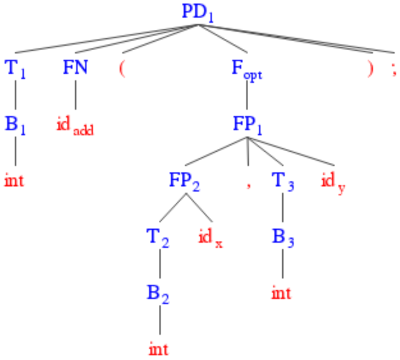
ST.gbl: ST.gbl.parent = null					
x	int	global	4	0	null
ar	array(2, array(3, int))				
		global	24	4	null
y	int	global	4	28	null

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 $\times$ int $\rightarrow$ int	func	0	32	ST.add()

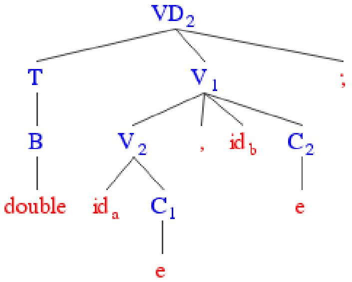
Columns: Name, Type, Category, Size, Offset, & Symtab

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
```



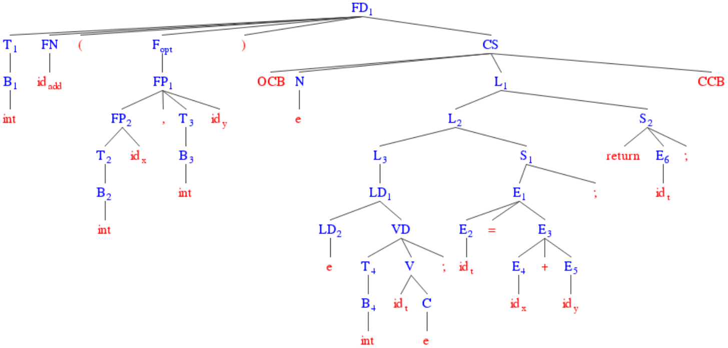

```
double a, b;              // VD_2
```

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 $\times$ int $\rightarrow$ 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)



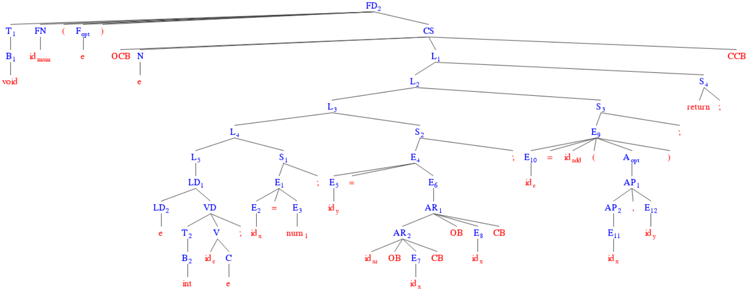
ST.gbl: ST.gbl.parent = null					
x	int	global	4	0	null
ar	array(2, array(3, int))				
		global	24	4	null
y	int	global	4	28	null
add	int $\times$ int $\rightarrow$ int				
		func	0	32	ST.add()
a	double	global	8	32	null
b	double	global	8	40	null

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

ST.add(): ST.add.parent = ST.gbl				
x	int	param	4	0
y	int	param	4	4
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)



$ST.gbl$: $ST.gbl.parent = null$					
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

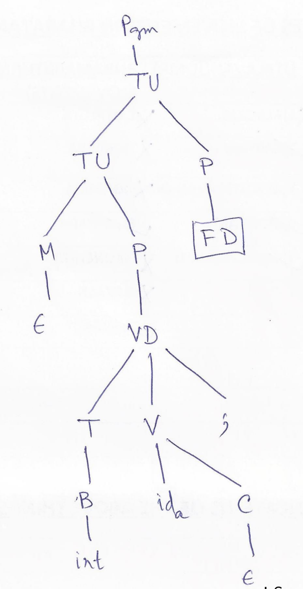
ST.main(): ST.main.parent = ST.gbl

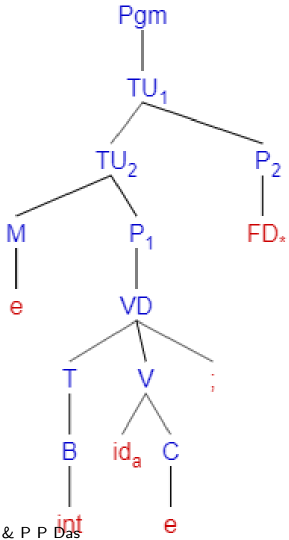
c	int	local	4	0
t#1	int	temp	4	4
t#2	int	temp	4	8
t#3	int	temp	4	12
t#4	int	temp	4	16

Example 2: Nested Blocks: Source

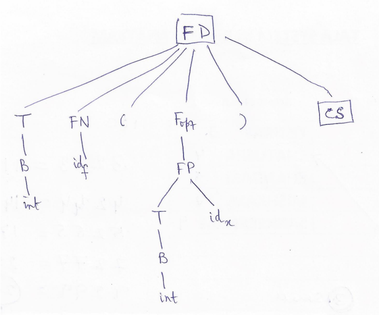
```
int a;
int f(int x) { // function scope f
    int t, u;
    t = x; // t in f, x in f
    { // un-named block scope f_1
        int p, q, t;
        p = a; // p in f_1, a in global
        t = 4; // t in f_1, hides t in f
        { // un-named block scope f_1_1
            int p;
            p = 5;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

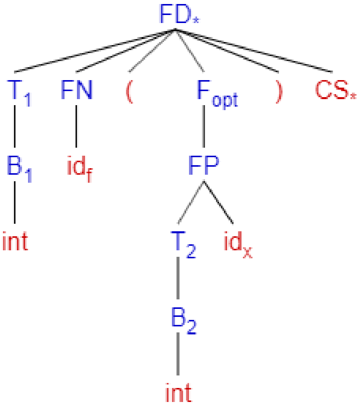
Example 2: Nested Blocks: Parse Tree (Pgm)



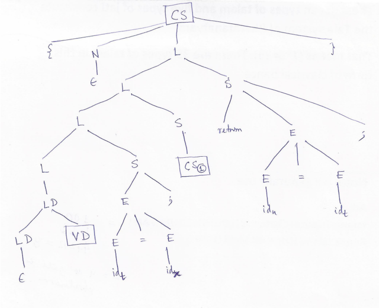


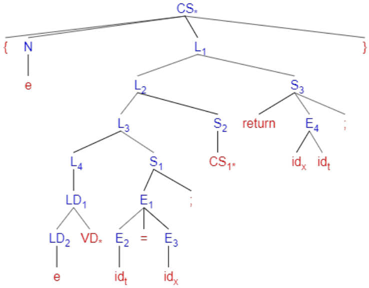
Example 2: Nested Blocks: Parse Tree (FD)



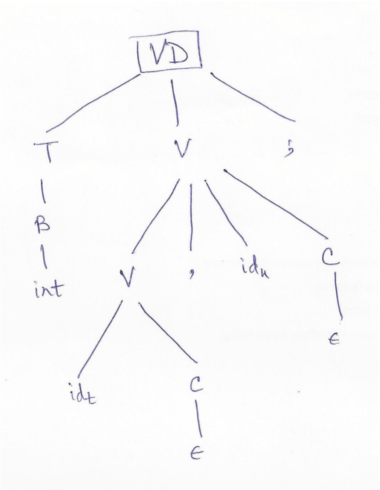


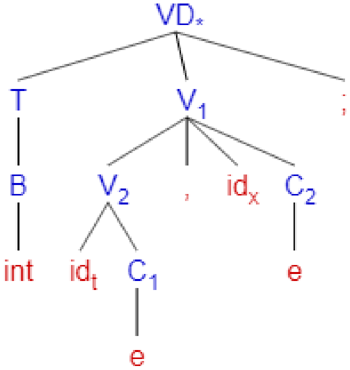
Example 2: Nested Blocks: Parse Tree (CS)



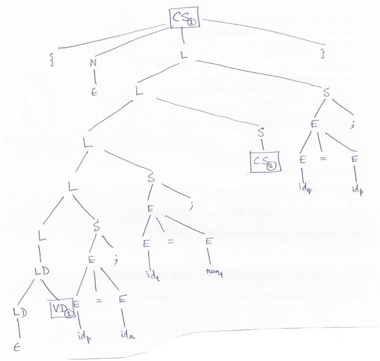


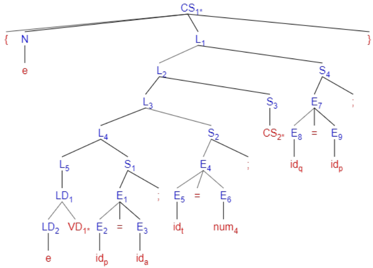
Example 2: Nested Blocks: Parse Tree (VD)



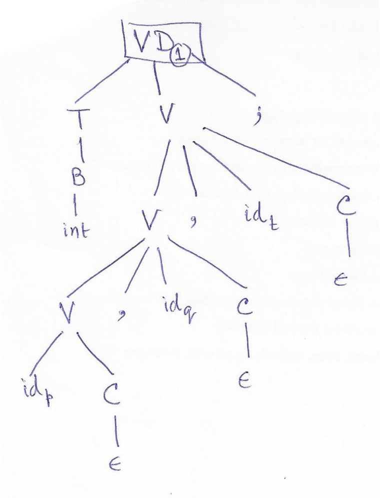


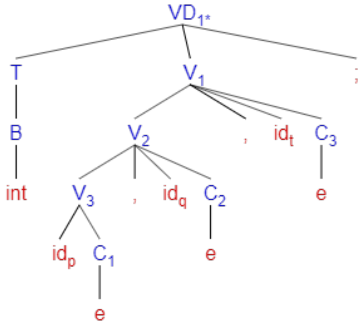
Example 2: Nested Blocks: Parse Tree (CS_1)



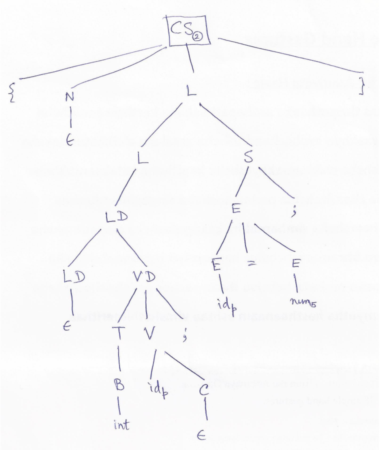


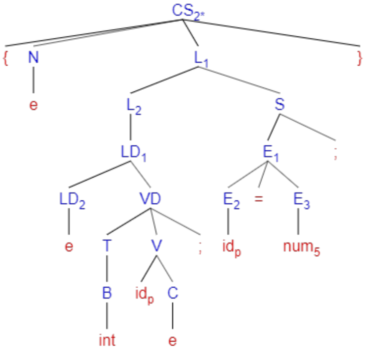
Example 2: Nested Blocks: Parse Tree (VD_1)



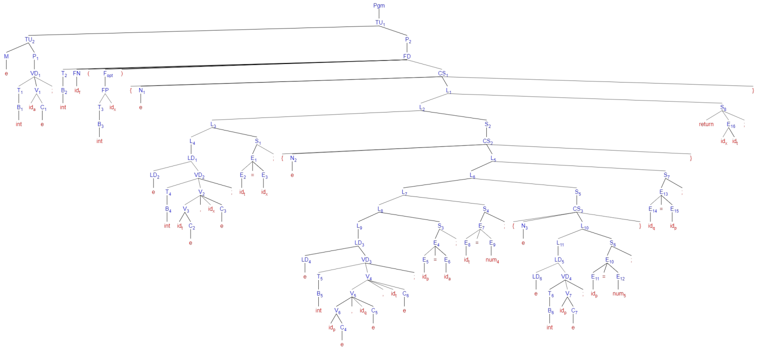


Example 2: Nested Blocks: Parse Tree (CS_2)





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



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
```

```
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
f_1	null	block	–		ST.f_1

ST.f_1: ST.f_1.parent = ST.f					
p	int	local	4	0	null
q	int	local	4	4	null
t	int	local	4	8	null
f_1_1	null	block	–		ST.f_1_1

ST.f_1_1: ST.f_1_1.parent = ST.f_1					
p	int	local	4	0	null

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

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

Additional Features

Additional Features

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