Basic Semantics

CSE340 FALL 2021

Rida Bazzi

Names and Attributes

In a programming language, we typically use names to refer to languages constructs. Language constructs also have attributes. We list constructs that can have names and possible attributes

Name	Attributes
variable	value
function	type
type	location
constant	size
scope	code
class	alignment
parameter	
module	
package	
label	
field	
macro	

Name have attributes associated with them. The association between a name and an attribute is called **binding**

Binding can be established at different times:

1. language definition time: when the language is defined.

   Example: boolean value true and false

2. language implementation time: for attributes that are implementation-dependent

   Example: value of MAX_INT which is implementation dependent

3. compile time: the association is done when the program is compiled

   Example: the type of x in the declaration int x;

4. link time: the association is done when the program is linked

   Example: the address of x in the declaration extern int x;

5. load time: the association is done when the program is loaded

   Example: the absolute address of a variable

6. runtime: The association is determined at runtime

   Example: the value of x in the C assignment x = y + z; (assuming compiler optimization does not determine the value at compile time)

   Example: the type of a in the Python assignment a = "abc";

Declarations and References

Names are introduced in a program by declaring them.

A ***declaration*** can be explicit or implicit. An explicit declaration introduces a name together with some attributes and is identified as such in the language syntax.

Examples

Explicit:	int x;	// introduced the name x
Implicit:	a = 5;	// in Python introduces the name a
		// without the need for a separate
		// explicit declaration
Implicit:	label_name :	// implicitly declared label
		// in C

Definition. A ***reference or a use*** of a name is a construct that contains a name that is separately or implicitly declared

Examples

In C:	int x;	// declares name x
		// x is name of a variable
		// location attribute associated with x
		// type attribute int associated with x
	x = 7;	// use or reference to x
In Python:	x = 7;	// implicit declaration of x
		// use of name x
	y = x;	// declares and uses y
		// use/reference of name x

resolving the reference

resolving the reference

Resolving a Reference and Scopes

Definition: ***Resolving a reference*** consists of determining the declaration that corresponds to a particular reference.

Definition: ***The scope of a specific declaration of name x*** is the region of the the program text (for static scoping) or the program execution (for dynamic scoping) in which a reference to x resolves to the specific declaration of x

The following example show declarations and identifies their scopes. I am using color coding to associate a use with a declaration. This example is not unlike what we have seen with lambda calculus

```
{
    int x;    // declaration 1
    int y;    // declaration 2

    x = 4;

    {
        int x;    // declaration 3
        int z;    // declaration 4
        y = 3;
        x = 5;
    }
    x = 6;

    y = 3;
    z = 7;    // invalid reference
}
```

scope x

scope y

scope x

scope z

Syntactic Constructs for Scopes

In many programming languages (C, Java, C++, Ada, ...) there is a non-terminal in the grammar of the language to define scopes. Typically, it is called "block".

Here we are talking about scopes independently of declaration (on the previous page we talked about scope of a declaration). The two concepts are not unrelated as we will see.

Example:

In C99, we have the following grammar fragments for blocks

compound-statement	→	{ block-item-list }
block-item-list	→	block-item
block-item-list	→	block-item-list block-item
block-item	→	declaration
block-item	→	statement

C99 divides scope into 4 kinds: (1) block scope, (2) file scope (global scope) (3) function scope for labels, and function prototype scope (for function declaration).

The block scope is the same as compound-statement.

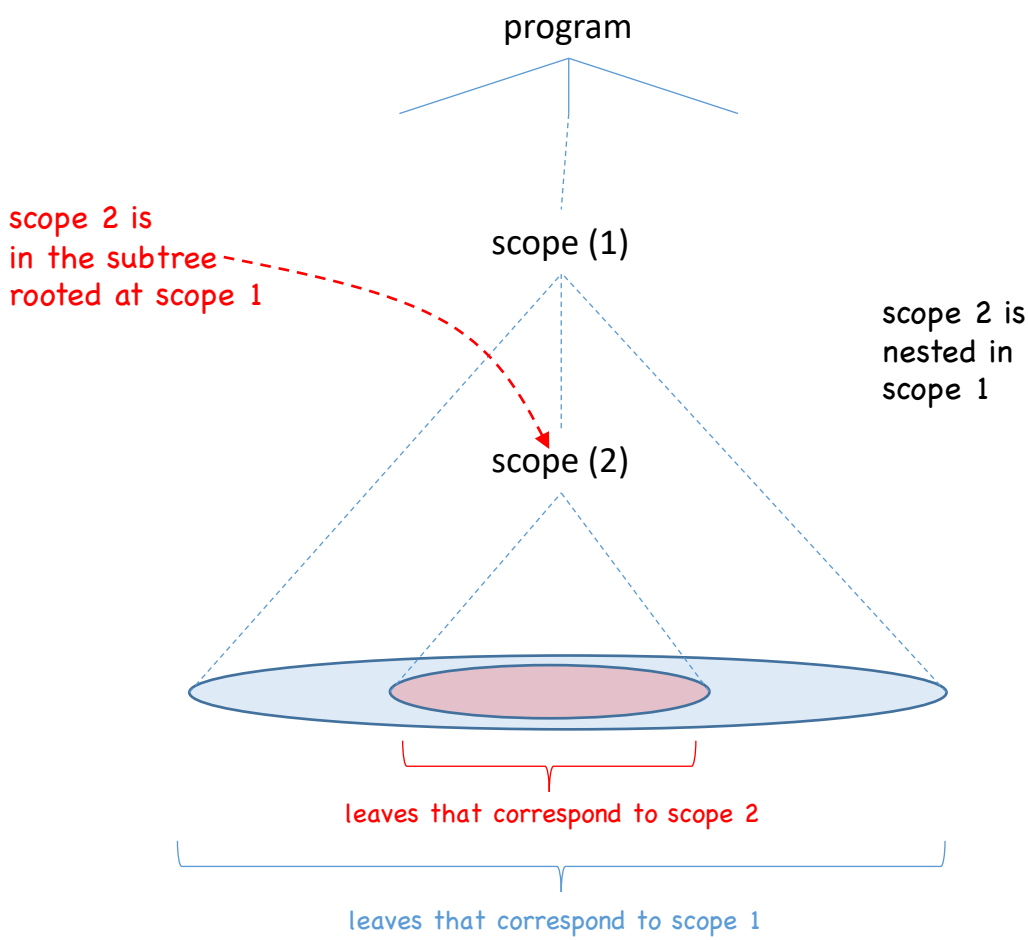
Nested scopes

We restrict our discussion to blocks (compound statements).

Definition: We say that block A is nested in block B if the parse tree node for block A is a descendent of the parse tree node for block B

Definition: We say that block A is immediately (or directly) nested in block B is

- A is nested in B and
- for every block C in which A is nested, block B is also nested in block C (think about this. It is not immediately obvious what it is saying!)



Nested scopes

<u>Example</u>: In the following example A is immediately nested in B. B and D are immediately nested in C, but A is not immediately nested in C. Blocks B and D are non-overlapping (neither is nested in the other).

```
{
        int x;      // declaration 1
        int y;      // declaration 2
        x = 4;
        {
                int x;      // declaration 3
                int z;      // declaration 4
                {
                        y = 3;
                        x = 5;
                }
        }

        {
                int z;
                z = x;
        }

        x = 6;
        y = 3;
}
```

B

A

C

D

Resolving a reference

We define a procedure lookup() to look up the declaration corresponding to a particular use. The pseudocode is the following:

local scope to start the lookup at

name to lookup

function to do local lookup amongst the local declarations in "scope"

```
lookup(scope , name) {
      if scope != NULL {
            if (lookup_in_local_scope(scope , name) != NULL)
                  return lookup_in_local_scope(scope,name);
            else
                  return lookup(scope->parent , name);
      } else
            return NULL
  }
```

recursive call

starting at parent scope

In the pseudocode, lookup() first attempts to lookup the name locally in the given scope (the scope parameter). If local lookup in the given scope fails, lookup is done recursively by calling the lookup function with the parent scope as the first argument

The pseudocode does not specify the type of the value returned by lookup. In general, it is a pointer to a declaration node (the specifics are implementation dependent). The code assume that the value returned is a pointer to a structure.

Finally, the pseudocode does not define what a parent scope is. It assumes that there is a way to refer to the parent scope of a given scope. In the code, this is done using scope->parent. We will define next what the parent scope is for both static and dynamic scoping.

Note Some languages provide ways to refer to names outside a given scope. For example

- Java: this.x // field of the current class
- C++: ::x // global x

Static and Dynamic Scoping

Definitions. Static and dynamic scoping.

In ***static scoping***, the parent scope is the directly enclosing scope. Static scoping is also called lexical scoping.

In ***dynamic scoping***

- within a function: the same as static scoping
- when a function is called: the caller is the parent of the callee (the called function)

The lookup function is the same for static and for dynamic scoping; the difference is in the definition of parent scope.

Note static scoping is what you are already familiar with from C++ or Java

Definition. A symbol table is a data structure used in resolving references and to keep information about attributes of names

Languages that use static scoping: almost any language you can think of: *C, C++ , Java, Modula, C#, Ada, ML*

Languages that support dynamic scoping: *Perl* and some variants of *Common Lisp* support dynamic scoping. They allow declarations that are dynamically scoped.

I give examples of dynamic and lexical scoping on the next couple pages

Example Static Scoping

Code

```
int a;

f()
{   int a = 1;
    g();
}

g()
{
   print a;
}

main()
{
   a = 3;
   f();
   g();
}
```

Execution trace

a : 3 (1)

main()

(1) a = 3
(2) f()

a : 1 (3)

(3) a = 1
(4) g()
(5) print a // prints 3

(6) g()
(7) print a // prints 3

Example Dynamic Scoping

Code

```
int a;

f()
{   int a = 1;
    g();
}

g()
{
   print a;
}

main()
{
   a = 3;
   f();
   g();
}
```

Execution trace

a : 3 (1)

main()

(1) a = 3
(2) f()

a : 1 (3)

(3) a = 1
(4) g()
(5) print a // prints 1

(6) g()
(7) print a // prints 3

Example Dynamic Scoping (Exam Fall 2014)

Code

```
1:  int a, b, c;

2:  int g(int x)
3:  {
4:     printf("%d %d %d\n", a, b, c);
5:     return a+b+c+x;
6:  }

7:  int f(int a)
8:  {      int b;
9:         b = 1;
10:        c = 2;
11:        {      int b;
12:               b = 9;
13:               a = 4;
14:               c = b+c;
15:               {      int a;
16:                      a = 3;
17:                      a = g(1);
18:                      printf("%d %d %d\n", a, b, c);
19:               }
20:        }
21:        {      // inner  scope 2
22:               int m;
23:               int n;
24:               m = g(3));
25:               printf("%d %d %d\n", m, a, b, c);
26:        }
27:        return a+b+c;
28: }

29: main()
30: {
31:     int b;
32:     int c;
33:     a = 11;
34:     b = 2;
35:     a = f(a);
36:     printf("%d %d %d\n", a, b, c);
37: }
```

Execution trace

a : 11 (1)
b:
c:

main()

b: 2 (2)
c: 2 (5) 11 (8)

(1) a = 11
(2) b = 2
a = f(a) = f(11)
(3) f(11)

a: 11 (argument) 4 (7)
b: 1 (4)

(4) b = 1
(5) c = 2

b : 9 (6)

(6) b = 9
(7) a = 4
(8) c = b+c = 9+2 = 11

a : 3 (9) 24 (10)

(9) a = 3
a = g(1)
g(1)

x : 1 (argument)

print a, b, c // 3,9,11
return a+b+c+x = 24

(10) a = g(1) = 24
print a, b, c // 24,9,11

. . .

Example Static Scoping Symbol Table

This example shows how the symbol table is built while parsing. Each step (between two lines shows the symbol table after various parts of he program are parsed.

Initially current_scope = NULL

nothing is parsed

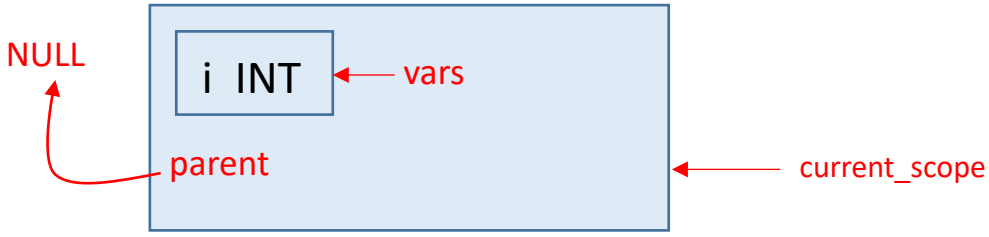
Example Static Scoping Symbol Table

{

A new scope is entered so a node is created for it and it becomes the current scope

Example Static Scoping Symbol Table

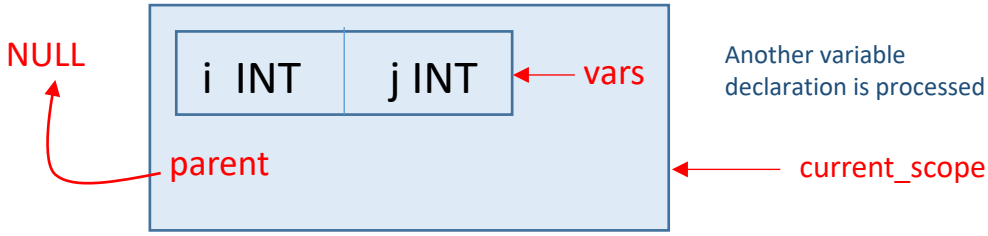
```
{  int i;
```



variable declaration is parsed and the information about the variable is inserted in current scope

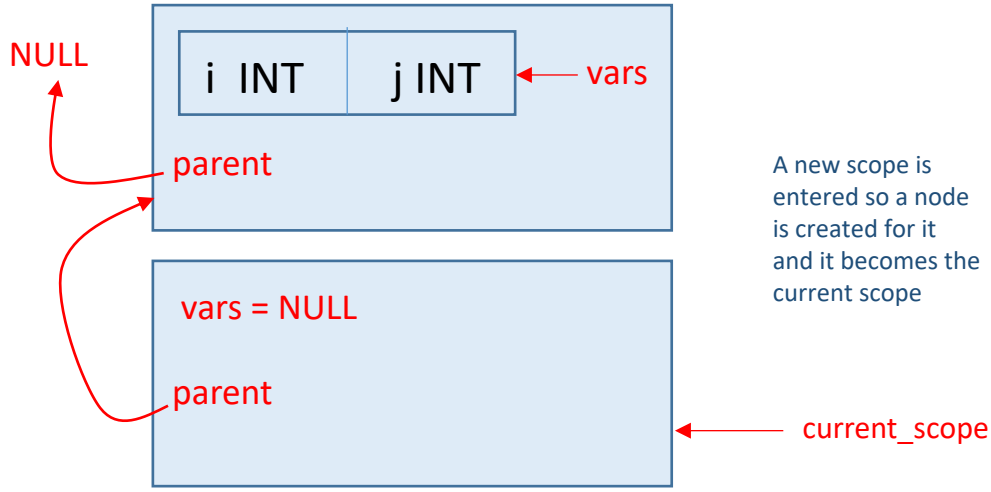
Example Static Scoping Symbol Table

```
{  int i;
   int j;
```



Example Static Scoping Symbol Table

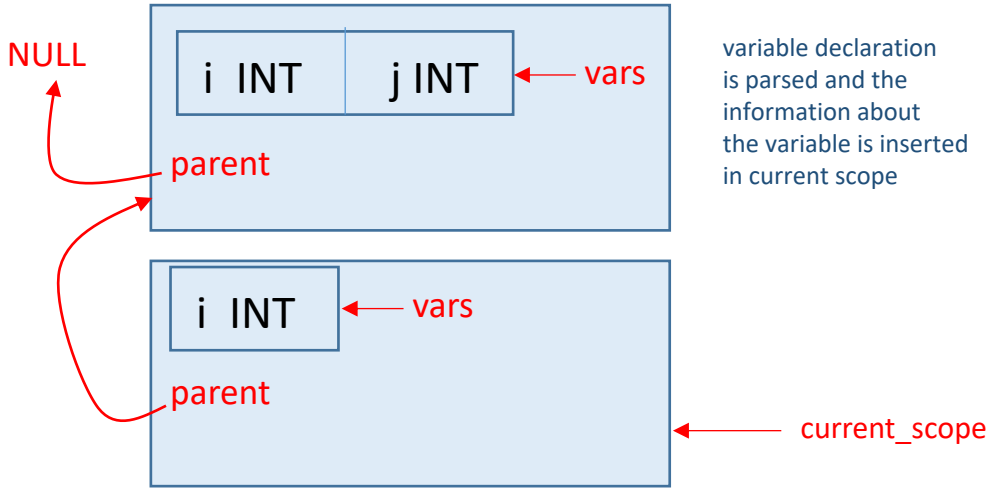
```
{  int i;
   int j;

   {
```



A new scope is entered so a node is created for it and it becomes the current scope

Example Static Scoping Symbol Table

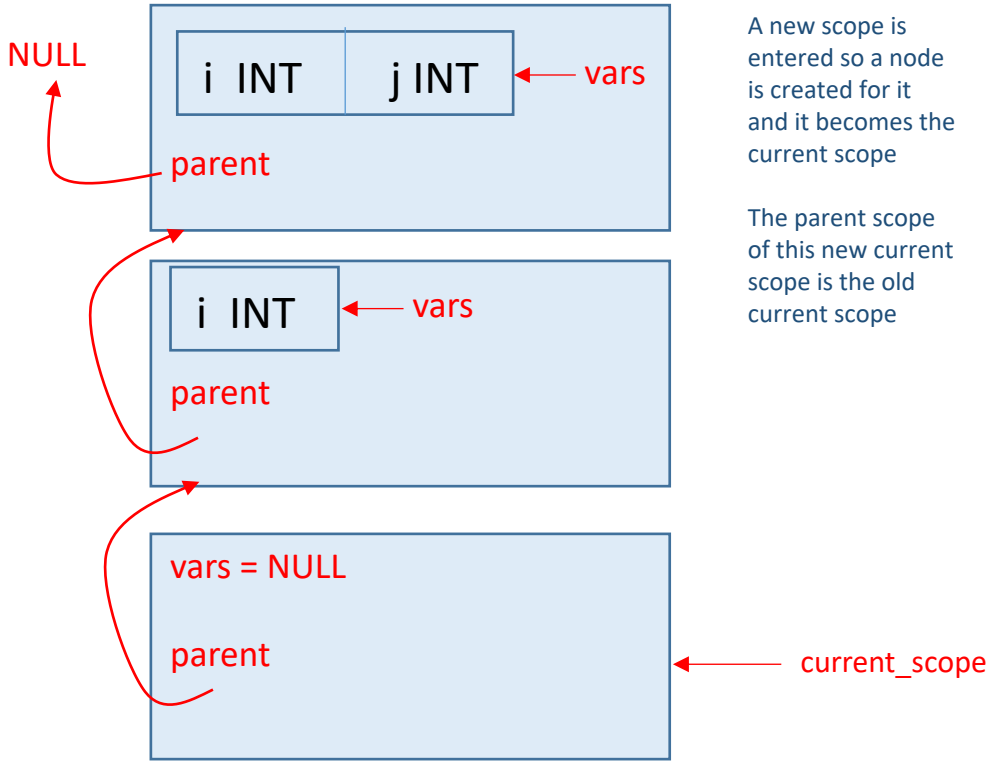
```
{  int i;
   int j;

   {  int i;
```



variable declaration is parsed and the information about the variable is inserted in current scope

Example Static Scoping Symbol Table

```
{  int i;
   int j;

   {  int i;

         {
```

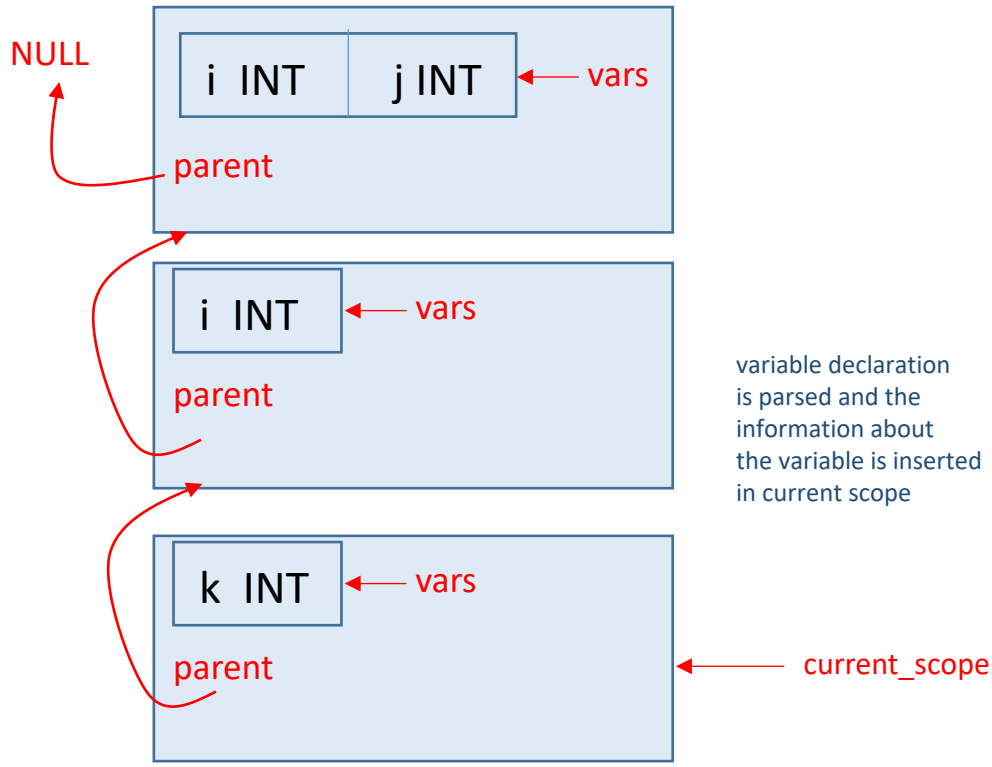


A new scope is entered so a node is created for it and it becomes the current scope

The parent scope of this new current scope is the old current scope

Example Static Scoping Symbol Table

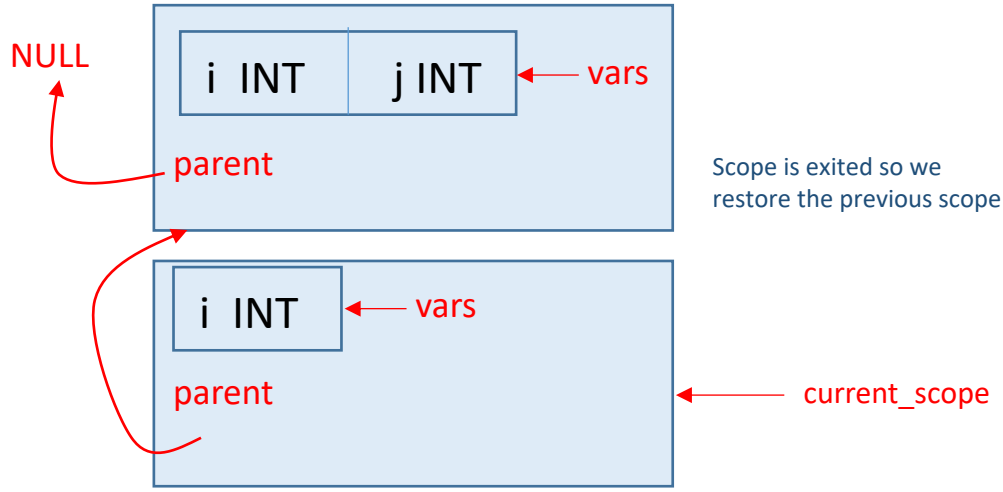
```
{  int i;
   int j;

   {  int i;

         { int k;
```



variable declaration is parsed and the information about the variable is inserted in current scope

Example Static Scoping Symbol Table

```
{  int i;
   int j;

   {  int i;

         { int k;


         }
```



How to build symbol table as you parse?

Initially current_scope = NULL

```
parse_scope()
{
    new_scope = new scope;     // allocates new scope structure
    new_scope->parent = current_scope;
    current_scope = new_scope;

       // parse the
       // rest of the
       // scope

   temp = current_scope;
   current_scope = current_scope->parent;
   delete temp;                        // C++ to free allocated memory
}
```

pseudocode to build Symbol Table
while parsing the input program

Note: **I only show code related to the symbol table. I omit all getToken() and checking for token type**

How to handle variable declarations?

Grammar rules for declarations

```
decl          -> id_list COLON type_name SEMICOLON
type_name  ->  INT | REAL | STRING | BOOLEAN
```

Grammar rules for declarations

```
decl          -> id_list COLON type_name SEMICOLON
type_name  ->  INT | REAL | STRING | BOOLEAN

parse_decl()
{
```

Grammar rules for declarations

```
decl          -> id_list COLON type_name SEMICOLON
type_name  ->  INT | REAL | STRING | BOOLEAN

parse_decl()
{
```

Example: i , j : INT;

Grammar rules for declarations

```
decl          -> id_list COLON type_name SEMICOLON
type_name  ->  INT | REAL | STRING | BOOLEAN
```

```
parse_decl()
{
    idList = parse_id_list();
```

Example: i , j : INT;

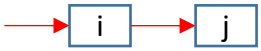
Grammar rules for declarations

```
decl          -> id_list COLON type_name SEMICOLON
type_name  ->  INT | REAL | STRING | BOOLEAN
```

```
parse_decl()
{
    idList = parse_id_list();
```

Example: i , j : INT;

idList



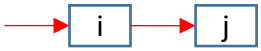
Grammar rules for declarations

```
decl          -> id_list COLON type_name SEMICOLON
type_name  ->  INT | REAL | STRING | BOOLEAN
```

```
parse_decl()
{
    idList = parse_id_list();
    typeName = parse_type_name();
```

Example: i , j : INT;

idList

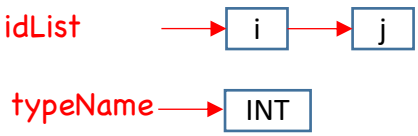


Grammar rules for declarations

```
decl          -> id_list COLON type_name SEMICOLON
type_name  ->  INT | REAL | STRING | BOOLEAN
```

```
parse_decl()
{
    idList = parse_id_list();
    typeName = parse_type_name();
```

Example: i , j : INT;

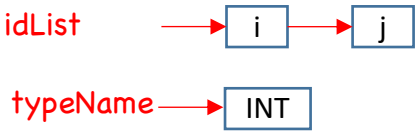


Grammar rules for declarations

```
decl          -> id_list COLON type_name SEMICOLON
type_name  ->  INT | REAL | STRING | BOOLEAN
```

```
parse_decl()
{
    idList = parse_id_list();
    typeName = parse_type_name();

    for each identifier in idList do
    {
```

Example: i , j : INT;

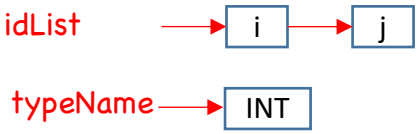


Grammar rules for declarations

```
decl          -> id_list COLON type_name SEMICOLON
type_name  ->  INT | REAL | STRING | BOOLEAN
```

```
parse_decl()
{
    idList = parse_id_list();
    typeName = parse_type_name();

    for each identifier in idList do
    {
            if identifier already declared in current scope then
            {
```

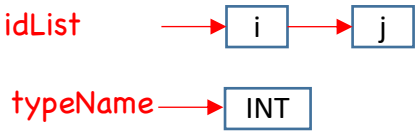
Example: i , j : INT;



Grammar rules for declarations

```
decl          -> id_list COLON type_name SEMICOLON
type_name  ->  INT | REAL | STRING | BOOLEAN
```

Example: i , j : INT;

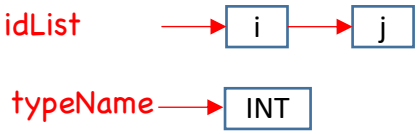



```
parse_decl()
{
    idList = parse_id_list();
    typeName = parse_type_name();

    for each identifier in idList do
    {
            if identifier already declared in current scope then
            {
                    error
            }
```

Grammar rules for declarations

```
decl          -> id_list COLON type_name SEMICOLON
type_name  ->  INT | REAL | STRING | BOOLEAN
```

Example: i , j : INT;

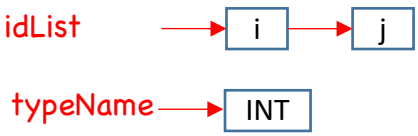



```
parse_decl()
{
    idList = parse_id_list();
    typeName = parse_type_name();

    for each identifier in idList do
    {
            if identifier already declared in current scope then
            {
                    error
            }
            else
```

Grammar rules for declarations

```
decl          -> id_list COLON type_name SEMICOLON
type_name  ->  INT | REAL | STRING | BOOLEAN
```

Example: i , j : INT;

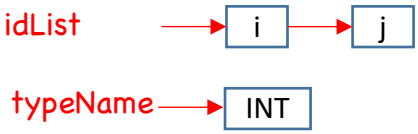



```
parse_decl()
{
    idList = parse_id_list();
    typeName = parse_type_name();

    for each identifier in idList do
    {
            if identifier already declared in current scope then
            {
                    error
            }
            else
            {
                    insert (identifier, typeName) in local
                    list of identifiers
```

Grammar rules for declarations

```
decl          -> id_list COLON type_name SEMICOLON
type_name  ->  INT | REAL | STRING | BOOLEAN
```

Example: i , j : INT;

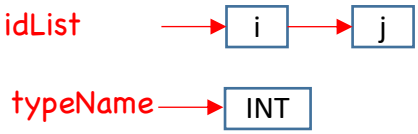



```
parse_decl()
{
    idList = parse_id_list();
    typeName = parse_type_name();

    for each identifier in idList do
    {
            if identifier already declared in current scope then
            {
                    error
            }
            else
            {
                    insert (identifier, typeName) in local
                    list of identifiers (I call it vars in the
                    illustrations on pages 14 and 15 above)
            }
```

Grammar rules for declarations

```
decl          -> id_list COLON type_name SEMICOLON
type_name  ->  INT | REAL | STRING | BOOLEAN
```

Example: i , j : INT;

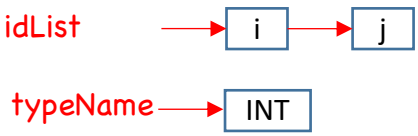



```
parse_decl()
{
    idList = parse_id_list();
    typeName = parse_type_name();

    for each identifier in idList do
    {
            if identifier already declared in current scope then
            {
                    error
            }
            else
            {
                    insert (identifier, typeName) in local
                    list of identifiers (I call it vars in the
                    illustrations on pages 14 and 15 above)
            }
    }
}
```

Grammar rules for declarations

```
decl          -> id_list COLON type_name SEMICOLON
type_name  ->  INT | REAL | STRING | BOOLEAN
```

Example: i , j : INT;



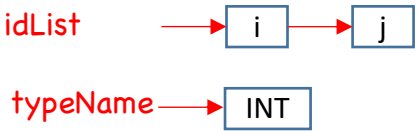

```
parse_decl()
{
    idList = parse_id_list();
    typeName = parse_type_name();

    for each identifier in idList do
    {
            if identifier already declared in current scope then
            {
                    error
            }
            else
            {
                    insert (identifier, typeName) in local
                    list of identifiers (I call it vars in the
                    illustrations on pages 14 and 15 above)
            }
    }
}
```

Note: I only show code/pseudocode related to the symbol table.
I omit all getToken() and checking for token type

Grammar rules for declarations

```
decl          -> id_list COLON type_name SEMICOLON
type_name  ->  INT | REAL | STRING | BOOLEAN
```

Example: i , j : INT;




```
parse_decl()
{
    idList = parse_id_list();
    typeName = parse_type_name();

    for each identifier in idList do
    {
            if identifier already declared in current scope then
            {
                    error
            }
            else
            {
                    insert (identifier, typeName) in local
                    list of identifiers (I call it vars in the
                    illustrations on pages 14 and 15 above)
            }
    }
}
```

Note: I only show code/pseudocode related to the symbol table.
I omit all getToken() and checking for token type

Note: The grammar might be different from the project grammar
but the concepts are the same

type checking for expressions

Example Grammar for Expressions

```
expr        -> operator expr expr
expr        -> primary
primary     ->  ID | NUM | REALNUM | BOOL_CONSTANT | STRING_CONST
type_name   ->  INT | REAL | STRING | BOOLEAN
```

```
expr        -> operator expr expr
expr        -> primary
primary     ->  ID | NUM | REALNUM | BOOL_CONSTANT | STRING_CONST
type_name   ->  INT | REAL | STRING | BOOLEAN
```

We are going to have parse_expr() and parse_primary() return a type which we represent as an integer

```
expr        -> operator expr expr
expr        -> primary
primary     ->  ID | NUM | REALNUM | BOOL_CONSTANT | STRING_CONST
type_name   ->  INT | REAL | STRING | BOOLEAN
```

We are going to have parse_expr() and parse_primary() return a type which we represent as an integer

```
int pare_primary()
{









}


int parse_expr()
{









}
```

```
expr        -> operator expr expr
expr        -> primary
primary     ->  ID | NUM | REALNUM | BOOL_CONSTANT | STRING_CONST
type_name   ->  INT | REAL | STRING | BOOLEAN
```

We are going to have parse_expr() and parse_primary() return a type which we represent as an integer

```
int paras_primary()
{
    case ID:





}


int parse_expr()
{





}
```

```
expr        -> operator expr expr
expr        -> primary
primary     ->  ID | NUM | REALNUM | BOOL_CONSTANT | STRING_CONST
type_name   ->  INT | REAL | STRING | BOOLEAN
```

We are going to have parse_expr() and parse_primary() return a type which we represent as an integer

```
int paras_primary()
{
    case ID:                    type = lookup_type(t.lexeme, current_scope)





}


int parse_expr()
{





}
```

```
expr        -> operator expr expr
expr        -> primary
primary     ->  ID | NUM | REALNUM | BOOL_CONSTANT | STRING_CONST
type_name   ->  INT | REAL | STRING | BOOLEAN
```

We are going to have parse_expr() and parse_primary() return a type which we represent as an integer

```
int parse_primary()
{
    case ID:                    type = lookup_type(t.lexeme, current_scope)
    case INT:





}


int parse_expr()
{





}
```

```
expr        -> operator expr expr
expr        -> primary
primary     ->  ID | NUM | REALNUM | BOOL_CONSTANT | STRING_CONST
type_name   ->  INT | REAL | STRING | BOOLEAN
```

We are going to have parse_expr() and parse_primary() return a type which we represent as an integer

```
int parse_primary()
{
    case ID:                type = lookup_type(t.lexeme, current_scope)
    case INT:               type = INT





}


int parse_expr()
{





}
```

```
expr        -> operator expr expr
expr        -> primary
primary     ->  ID | NUM | REALNUM | BOOL_CONSTANT | STRING_CONST
type_name   ->  INT | REAL | STRING | BOOLEAN
```

We are going to have parse_expr() and parse_primary() return a type which we represent as an integer

```
int parse_primary()
{
    case ID:                type = lookup_type(t.lexeme, current_scope)
    case INT:               type = INT
    case REALNUM:





}


int parse_expr()
{





}
```

```
expr        -> operator expr expr
expr        -> primary
primary     ->  ID | NUM | REALNUM | BOOL_CONSTANT | STRING_CONST
type_name   ->  INT | REAL | STRING | BOOLEAN
```

We are going to have parse_expr() and parse_primary() return a type which we represent as an integer

```
int parse_primary()
{
    case ID:                type = lookup_type(t.lexeme, current_scope)
    case INT:               type = INT
    case REALNUM:           type = REAL





}


int parse_expr()
{





}
```

```
expr        -> operator expr expr
expr        -> primary
primary     ->  ID | NUM | REALNUM | BOOL_CONSTANT | STRING_CONST
type_name   ->  INT | REAL | STRING | BOOLEAN
```

We are going to have parse_expr() and parse_primary() return a type which we represent as an integer

```
int parse_primary()
{
    case ID:                type = lookup_type(t.lexeme, current_scope)
    case INT:               type = INT
    case REALNUM:           type = REAL
        .
        .
        .
    return type;
}


int parse_expr()
{





}
```

```
expr        -> operator expr expr
expr        -> primary
primary     ->  ID | NUM | REALNUM | BOOL_CONSTANT | STRING_CONST
type_name   ->  INT | REAL | STRING | BOOLEAN
```

We are going to have parse_expr() and parse_primary() return a type which we represent as an integer

```
int parse_primary()
{
    case ID:                type = lookup_type(t.lexeme, current_scope)
    case INT:               type = INT
    case REALNUM:           type = REAL
        .
        .
        .
    return type;
}


int parse_expr()
{
     case primary :




}
```

```
expr        -> operator expr expr
expr        -> primary
primary     ->  ID | NUM | REALNUM | BOOL_CONSTANT | STRING_CONST
type_name   ->  INT | REAL | STRING | BOOLEAN
```

We are going to have parse_expr() and parse_primary() return a type which we represent as an integer

```
int parse_primary()
{
    case ID:                type = lookup_type(t.lexeme, current_scope)
    case INT:               type = INT
    case REALNUM:           type = REAL
        .
        .
        .
    return type;
}


int parse_expr()
{
     case primary :                     type = parse_primary()





}
```

```
expr        -> operator expr expr
expr        -> primary
primary     ->  ID | NUM | REALNUM | BOOL_CONSTANT | STRING_CONST
type_name   ->  INT | REAL | STRING | BOOLEAN
```

We are going to have parse_expr() and parse_primary() return a type which we represent as an integer

```
int parse_primary()
{
    case ID:                type = lookup_type(t.lexeme, current_scope)
    case INT:               type = INT
    case REALNUM:           type = REAL
        .
        .
        .
    return type;
}


int parse_expr()
{
     case primary :                     type = parse_primary()
     case operator expr expr:




}
```

How to handle errors for project 3:

if there is an error, type_check() should save the error message, but we need to avoid having more error messages. You can have type check with a flag that is set once an error is detected so that later errors detected by type_check() do not produce error messages

```
expr        -> operator expr expr
expr        -> primary
primary     ->  ID | NUM | REALNUM | BOOL_CONSTANT | STRING_CONST
type_name   ->  INT | REAL | STRING | BOOLEAN
```

We are going to have parse_expr() and parse_primary() return a type which we represent as an integer

```
int parse_primary()
{
    case ID:                type = lookup_type(t.lexeme, current_scope)
    case INT:               type = INT
    case REALNUM:           type = REAL
        .
        .
        .
    return type;
}


int parse_expr()
{
     case primary :                     type = parse_primary()
     case operator expr expr:           type1 = parse_expr()




}
```

```
expr        -> operator expr expr
expr        -> primary
primary     ->  ID | NUM | REALNUM | BOOL_CONSTANT | STRING_CONST
type_name   ->  INT | REAL | STRING | BOOLEAN
```

We are going to have parse_expr() and parse_primary() return a type which we represent as an integer

```
int parse_primary()
{
    case ID:                type = lookup_type(t.lexeme, current_scope)
    case INT:               type = INT
    case REALNUM:           type = REAL
        .
        .
        .
    return type;
}


int parse_expr()
{
     case primary :                     type = parse_primary()
     case operator expr expr:           type1 = parse_expr()
                                        type2 = parse_expr()



}
```

```
expr        -> operator expr expr
expr        -> primary
primary     ->  ID | NUM | REALNUM | BOOL_CONSTANT | STRING_CONST
type_name   ->  INT | REAL | STRING | BOOLEAN
```

We are going to have parse_expr() and parse_primary() return a type which we represent as an integer

```
int parse_primary()
{
    case ID:                type = lookup_type(t.lexeme, current_scope)
    case INT:               type = INT
    case REALNUM:           type = REAL
        .
        .
        .
    return type;
}


int parse_expr()
{
     case primary :                     type = parse_primary()
     case operator expr expr:           type1 = parse_expr
                                        type2 = parse_expr
                                        type = type_check(operator, type1, type2)


}
```

```
expr        -> operator expr expr
expr        -> primary
primary     ->  ID | NUM | REALNUM | BOOL_CONSTANT | STRING_CONST
type_name   ->  INT | REAL | STRING | BOOLEAN
```

We are going to have parse_expr() and parse_primary() return a type which we represent as an integer

```
int parse_primary()
{
    case ID:                type = lookup_type(t.lexeme, current_scope)
    case INT:               type = INT
    case REALNUM:           type = REAL
        .
        .
        .
    return type;
}


int parse_expr()
{
     case primary :                     type = parse_primary()
     case operator expr expr:           type1 = parse_expr
                                        type2 = parse_expr
                                        type = type_check(operator, type1, type2)

    return type;
}
```

```
expr        -> operator expr expr
expr        -> primary
primary     ->  ID | NUM | REALNUM | BOOL_CONSTANT | STRING_CONST
type_name   ->  INT | REAL | STRING | BOOLEAN
```

We are going to have parse_expr() and parse_primary() return a type which we represent as an integer

```
int parse_primary()
{
    case ID:                type = lookup_type(t.lexeme, current_scope)
    case INT:               type = INT
    case REALNUM:           type = REAL
        .
        .
        .
    return type;
}


int parse_expr()
{
     case primary :                     type = parse_primary()
     case operator expr expr:           type1 = parse_expr
                                        type2 = parse_expr
                                        type = type_check(operator, type1, type2)

    return type;
}
```

type_check() will apply the language rules to determine a type for he expression.

```
expr        -> operator expr expr
expr        -> primary
primary     ->  ID | NUM | REALNUM | BOOL_CONSTANT | STRING_CONST
type_name   ->  INT | REAL | STRING | BOOLEAN
```

We are going to have parse_expr() and parse_primary() return a type which we represent as an integer

```
int parse_primary()
{
    case ID:                type = lookup_type(t.lexeme, current_scope)
    case INT:               type = INT
    case REALNUM:           type = REAL
        .
        .
        .
    return type;
}
```

```
int parse_expr()
{
     case primary :                     type = parse_primary()
     case operator expr expr:           type1 = parse_expr
                                        type2 = parse_expr
                                        type = type_check(operator, type1, type2)

    return type;
}
```

type_check() will apply the language rules to determine a type for he expression.

For example,

```
expr        -> operator expr expr
expr        -> primary
primary     ->  ID | NUM | REALNUM | BOOL_CONSTANT | STRING_CONST
type_name   ->  INT | REAL | STRING | BOOLEAN
```

We are going to have parse_expr() and parse_primary() return a type which we represent as an integer

```
int parse_primary()
{
    case ID:                type = lookup_type(t.lexeme, current_scope)
    case INT:               type = INT
    case REALNUM:           type = REAL
        .
        .
        .
    return type;
}
```

```
int parse_expr()
{
     case primary :                     type = parse_primary()
     case operator expr expr:           type1 = parse_expr
                                        type2 = parse_expr
                                        type = type_check(operator, type1, type2)

    return type;
}
```

type_check() will apply the language rules to determine a type for he expression.

For example,

```
type1 = INT
type2 = INT
operator = DIV (division)
```

```
expr        -> operator expr expr
expr        -> primary
primary     ->  ID | NUM | REALNUM | BOOL_CONSTANT | STRING_CONST
type_name   ->  INT | REAL | STRING | BOOLEAN
```

We are going to have parse_expr() and parse_primary() return a type which we represent as an integer

```
int parse_primary()
{
    case ID:                type = lookup_type(t.lexeme, current_scope)
    case INT:               type = INT
    case REALNUM:           type = REAL
        .
        .
        .
    return type;
}
```

```
int parse_expr()
{
     case primary :                     type = parse_primary()
     case operator expr expr:           type1 = parse_expr
                                        type2 = parse_expr
                                        type = type_check(operator, type1, type2)

    return type;
}
```

type_check() will apply the language rules to determine a type for he expression.

For example,

```
type1 = INT
type2 = INT
operator = DIV (division)
```

the language rule might specify that the result if REAL

```
expr        -> operator expr expr
expr        -> primary
primary     ->  ID | NUM | REALNUM | BOOL_CONSTANT | STRING_CONST
type_name   ->  INT | REAL | STRING | BOOLEAN
```

We are going to have parse_expr() and parse_primary() return a type which we represent as an integer

```
int parse_primary()
{
    case ID:                type = lookup_type(t.lexeme, current_scope)
    case INT:               type = INT
    case REALNUM:           type = REAL
        .
        .
        .
    return type;
}


int parse_expr()
{
     case primary :                     type = parse_primary()
     case operator expr expr:           type1 = parse_expr
                                        type2 = parse_expr
                                        type = type_check(operator, type1, type2)

    return type;
}
```

How to handle errors?

```
expr        -> operator expr expr
expr        -> primary
primary     ->  ID | NUM | REALNUM | BOOL_CONSTANT | STRING_CONST
type_name   ->  INT | REAL | STRING | BOOLEAN
```

We are going to have parse_expr() and parse_primary() return a type which we represent as an integer

```
int parse_primary()
{
    case ID:                type = lookup_type(t.lexeme, current_scope)
    case INT:               type = INT
    case REALNUM:           type = REAL
        .
        .
        .
    return type;
}
```

```
int parse_expr()
{
     case primary :                     type = parse_primary()
     case operator expr expr:           type1 = parse_expr
                                        type2 = parse_expr
                                        type = type_check(operator, type1, type2)

    return type;
}
```

How to handle errors?

if there is an error, type_check() should can return ERROR and produce an error message.

```
expr        -> operator expr expr
expr        -> primary
primary     ->  ID | NUM | REALNUM | BOOL_CONSTANT | STRING_CONST
type_name   ->  INT | REAL | STRING | BOOLEAN
```

We are going to have parse_expr() and parse_primary() return a type which we represent as an integer

```
int parse_primary()
{
    case ID:                type = lookup_type(t.lexeme, current_scope)
    case INT:               type = INT
    case REALNUM:           type = REAL
        .
        .
        .
    return type;
}
```

```
int parse_expr()
{
     case primary :                     type = parse_primary()
     case operator expr expr:           type1 = parse_expr
                                        type2 = parse_expr
                                        type = type_check(operator, type1, type2)

    return type;
}
```

How to handle errors?

if there is an error, type_check() should can return ERROR and produce an error message.

If the requirements are that only one error message is produced, then a flag should be set to void more error messages.

```
expr        -> operator expr expr
expr        -> primary
primary     ->  ID | NUM | REALNUM | BOOL_CONSTANT | STRING_CONST
type_name   ->  INT | REAL | STRING | BOOLEAN
```

We are going to have parse_expr() and parse_primary() return a type which we represent as an integer

```
int parse_primary()
{
    case ID:                type = lookup_type(t.lexeme, current_scope)
    case INT:               type = INT
    case REALNUM:           type = REAL
        .
        .
        .
    return type;
}
```

```
int parse_expr()
{
     case primary :                     type = parse_primary()
     case operator expr expr:           type1 = parse_expr
                                        type2 = parse_expr
                                        type = type_check(operator, type1, type2)

    return type;
}
```

How to handle errors?

if there is an error, type_check() should can return ERROR and produce an error message.

If the requirements prioritize error messages, then the error messages need to be saved for later printing

type checking for expressions: Example Execution

+ + i j / i j

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                    // PLUS
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                    // PLUS
            type1 = parse_expr()
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                    // PLUS
            type1 = parse_expr()
                        operator = parse_operator();                        // PLUS
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                    // PLUS
            type1 = parse_expr()
                        operator = parse_operator();                        // PLUS
                        type1 = parse_expr()
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                    // PLUS
            type1 = parse_expr()
                        operator = parse_operator();                        // PLUS
                        type1 = parse_expr()
                                    type = parse_primary()                  // i
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                        operator = parse_operator();                      // PLUS
                        type1 = parse_expr()
                                    type = parse_primary()                // i
                                              type = lookup_type("i") = INT;
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                        operator = parse_operator();                      // PLUS
                        type1 = parse_expr()
                                    type = parse_primary()                // i
                                             type = lookup_type("i") = INT;
                                             return INT;
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                        operator = parse_operator();                      // PLUS
                        type1 = parse_expr()
                                    type = parse_primary()                // i
                                              type = lookup_type("i") = INT;
                                              return INT;
                              = INT
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                            operator = parse_operator();                  // PLUS
                            type1 = parse_expr()
                                        type = parse_primary()            // i
                                                  type = lookup_type("i") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                            operator = parse_operator();                  // PLUS
                            type1 = parse_expr()
                                        type = parse_primary()            // i
                                                  type = lookup_type("i") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // j
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                            operator = parse_operator();                  // PLUS
                            type1 = parse_expr()
                                        type = parse_primary()            // i
                                                  type = lookup_type("i") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // j
                                                  type = lookup_type("j") = INT;
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                            operator = parse_operator();                  // PLUS
                            type1 = parse_expr()
                                        type = parse_primary()            // i
                                                  type = lookup_type("i") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // j
                                                  type = lookup_type("j") = INT;
                                                  return INT;
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                            operator = parse_operator();                  // PLUS
                            type1 = parse_expr()
                                        type = parse_primary()            // i
                                                  type = lookup_type("i") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // j
                                                  type = lookup_type("j") = INT;
                                                  return INT;
                                  = INT
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                            operator = parse_operator();                  // PLUS
                            type1 = parse_expr()
                                        type = parse_primary()            // i
                                                  type = lookup_type("i") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // j
                                                  type = lookup_type("j") = INT;
                                                  return INT;
                                  = INT
                            type  =  type_check(operator, type1, type2)
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                        operator = parse_operator();                      // PLUS
                        type1 = parse_expr()
                                    type = parse_primary()                // i
                                              type = lookup_type("i") = INT;
                                              return INT;
                              = INT
                        type2 = parse_expr()
                                    type = parse_primary()                // j
                                              type = lookup_type("j") = INT;
                                              return INT;
                              = INT
                        type  =  type_check(operator, type1, type2)
                              =  type_check(PLUS, INT, INT) = INT;
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                        operator = parse_operator();                      // PLUS
                        type1 = parse_expr()
                                    type = parse_primary()                // i
                                              type = lookup_type("i") = INT;
                                              return INT;
                              = INT
                        type2 = parse_expr()
                                    type = parse_primary()                // j
                                              type = lookup_type("j") = INT;
                                              return INT;
                              = INT
                        type  =  type_check(operator, type1, type2)
                              =  type_check(PLUS, INT, INT) = INT;
                        return INT;
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                            operator = parse_operator();                  // PLUS
                            type1 = parse_expr()
                                        type = parse_primary()            // i
                                                  type = lookup_type("i") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // j
                                                  type = lookup_type("j") = INT;
                                                  return INT;
                                  = INT
                            type  =  type_check(operator, type1, type2)
                                  =  type_check(PLUS, INT, INT) = INT;
                            return INT;
            type1  = INT
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                            operator = parse_operator();                  // PLUS
                            type1 = parse_expr()
                                        type = parse_primary()            // i
                                                  type = lookup_type("i") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // j
                                                  type = lookup_type("j") = INT;
                                                  return INT;
                                  = INT
                            type  =  type_check(operator, type1, type2)
                                  =  type_check(PLUS, INT, INT) = INT;
                            return INT;
            type1  = INT

            type2 = parse_expr()
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                            operator = parse_operator();                  // PLUS
                            type1 = parse_expr()
                                        type = parse_primary()            // i
                                                  type = lookup_type("i") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // j
                                                  type = lookup_type("j") = INT;
                                                  return INT;
                                  = INT
                            type  =  type_check(operator, type1, type2)
                                  =  type_check(PLUS, INT, INT) = INT;
                            return INT;
            type1  = INT

            type2 = parse_expr()
                            operator = parse_operator();                  // DIV
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                            operator = parse_operator();                  // PLUS
                            type1 = parse_expr()
                                        type = parse_primary()            // i
                                                  type = lookup_type("i") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // j
                                                  type = lookup_type("j") = INT;
                                                  return INT;
                                  = INT
                            type  =  type_check(operator, type1, type2)
                                  =  type_check(PLUS, INT, INT) = INT;
                            return INT;
            type1  = INT

            type2 = parse_expr()
                            operator = parse_operator();                  // DIV
                            type1 = parse_expr()
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                            operator = parse_operator();                  // PLUS
                            type1 = parse_expr()
                                        type = parse_primary()            // i
                                                  type = lookup_type("i") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // j
                                                  type = lookup_type("j") = INT;
                                                  return INT;
                                  = INT
                            type  =  type_check(operator, type1, type2)
                                  =  type_check(PLUS, INT, INT) = INT;
                            return INT;
            type1  = INT

            type2 = parse_expr()
                            operator = parse_operator();                  // DIV
                            type1 = parse_expr()
                                        type = parse_primary()            // k
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                            operator = parse_operator();                  // PLUS
                            type1 = parse_expr()
                                        type = parse_primary()            // i
                                                  type = lookup_type("i") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // j
                                                  type = lookup_type("j") = INT;
                                                  return INT;
                                  = INT
                            type  =  type_check(operator, type1, type2)
                                  =  type_check(PLUS, INT, INT) = INT;
                            return INT;
            type1  = INT

            type2 = parse_expr()
                            operator = parse_operator();                  // DIV
                            type1 = parse_expr()
                                        type = parse_primary()            // k
                                                  type = lookup_type("k") = INT;
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                            operator = parse_operator();                  // PLUS
                            type1 = parse_expr()
                                        type = parse_primary()            // i
                                                  type = lookup_type("i") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // j
                                                  type = lookup_type("j") = INT;
                                                  return INT;
                                  = INT
                            type  =  type_check(operator, type1, type2)
                                  =  type_check(PLUS, INT, INT) = INT;
                            return INT;
            type1  = INT

            type2 = parse_expr()
                            operator = parse_operator();                  // DIV
                            type1 = parse_expr()
                                        type = parse_primary()            // k
                                                  type = lookup_type("k") = INT;
                                                  return INT;
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                            operator = parse_operator();                  // PLUS
                            type1 = parse_expr()
                                        type = parse_primary()            // i
                                                  type = lookup_type("i") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // j
                                                  type = lookup_type("j") = INT;
                                                  return INT;
                                  = INT
                            type  =  type_check(operator, type1, type2)
                                  =  type_check(PLUS, INT, INT) = INT;
                            return INT;
            type1  = INT

            type2 = parse_expr()
                            operator = parse_operator();                  // DIV
                            type1 = parse_expr()
                                        type = parse_primary()            // k
                                                  type = lookup_type("k") = INT;
                                                  return INT;
                                  = INT
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                            operator = parse_operator();                  // PLUS
                            type1 = parse_expr()
                                        type = parse_primary()            // i
                                                  type = lookup_type("i") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // j
                                                  type = lookup_type("j") = INT;
                                                  return INT;
                                  = INT
                            type  =  type_check(operator, type1, type2)
                                  =  type_check(PLUS, INT, INT) = INT;
                            return INT;
            type1  = INT

            type2 = parse_expr()
                            operator = parse_operator();                  // DIV
                            type1 = parse_expr()
                                        type = parse_primary()            // k
                                                  type = lookup_type("k") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                            operator = parse_operator();                  // PLUS
                            type1 = parse_expr()
                                        type = parse_primary()            // i
                                                  type = lookup_type("i") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // j
                                                  type = lookup_type("j") = INT;
                                                  return INT;
                                  = INT
                            type  =  type_check(operator, type1, type2)
                                  =  type_check(PLUS, INT, INT) = INT;
                            return INT;
            type1  = INT

            type2 = parse_expr()
                            operator = parse_operator();                  // DIV
                            type1 = parse_expr()
                                        type = parse_primary()            // k
                                                  type = lookup_type("k") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // l
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                            operator = parse_operator();                  // PLUS
                            type1 = parse_expr()
                                        type = parse_primary()            // i
                                                  type = lookup_type("i") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // j
                                                  type = lookup_type("j") = INT;
                                                  return INT;
                                  = INT
                            type  =  type_check(operator, type1, type2)
                                  =  type_check(PLUS, INT, INT) = INT;
                            return INT;
            type1  = INT

            type2 = parse_expr()
                            operator = parse_operator();                  // DIV
                            type1 = parse_expr()
                                        type = parse_primary()            // k
                                                  type = lookup_type("k") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // l
                                                  type = lookup_type("l") = INT;
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                            operator = parse_operator();                  // PLUS
                            type1 = parse_expr()
                                        type = parse_primary()            // i
                                                  type = lookup_type("i") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // j
                                                  type = lookup_type("j") = INT;
                                                  return INT;
                                  = INT
                            type  =  type_check(operator, type1, type2)
                                  =  type_check(PLUS, INT, INT) = INT;
                            return INT;
            type1  = INT

            type2 = parse_expr()
                            operator = parse_operator();                  // DIV
                            type1 = parse_expr()
                                        type = parse_primary()            // k
                                                  type = lookup_type("k") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // l
                                                  type = lookup_type("l") = INT;
                                                  return INT;
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                            operator = parse_operator();                  // PLUS
                            type1 = parse_expr()
                                        type = parse_primary()            // i
                                                  type = lookup_type("i") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // j
                                                  type = lookup_type("j") = INT;
                                                  return INT;
                                  = INT
                            type  =  type_check(operator, type1, type2)
                                  =  type_check(PLUS, INT, INT) = INT;
                            return INT;
            type1  = INT

            type2 = parse_expr()
                            operator = parse_operator();                  // DIV
                            type1 = parse_expr()
                                        type = parse_primary()            // k
                                                  type = lookup_type("k") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // l
                                                  type = lookup_type("l") = INT;
                                                  return INT;
                                  = INT
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                            operator = parse_operator();                  // PLUS
                            type1 = parse_expr()
                                        type = parse_primary()            // i
                                                  type = lookup_type("i") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // j
                                                  type = lookup_type("j") = INT;
                                                  return INT;
                                  = INT
                            type  =  type_check(operator, type1, type2)
                                  =  type_check(PLUS, INT, INT) = INT;
                            return INT;
            type1  = INT

            type2 = parse_expr()
                            operator = parse_operator();                  // DIV
                            type1 = parse_expr()
                                        type = parse_primary()            // k
                                                  type = lookup_type("k") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // l
                                                  type = lookup_type("l") = INT;
                                                  return INT;
                                  = INT
                            type  =  type_check(operator, type1, type2)
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                            operator = parse_operator();                  // PLUS
                            type1 = parse_expr()
                                        type = parse_primary()            // i
                                                  type = lookup_type("i") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // j
                                                  type = lookup_type("j") = INT;
                                                  return INT;
                                  = INT
                            type  =  type_check(operator, type1, type2)
                                  =  type_check(PLUS, INT, INT) = INT;
                            return INT;
            type1  = INT

            type2 = parse_expr()
                            operator = parse_operator();                  // DIV
                            type1 = parse_expr()
                                        type = parse_primary()            // k
                                                  type = lookup_type("k") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // l
                                                  type = lookup_type("l") = INT;
                                                  return INT;
                                  = INT
                            type  =  type_check(operator, type1, type2)
                                  =  type_check(PLUS, INT, INT) = INT;
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                            operator = parse_operator();                  // PLUS
                            type1 = parse_expr()
                                        type = parse_primary()            // i
                                                  type = lookup_type("i") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // j
                                                  type = lookup_type("j") = INT;
                                                  return INT;
                                  = INT
                            type  =  type_check(operator, type1, type2)
                                  =  type_check(PLUS, INT, INT) = INT;
                            return INT;
            type1  = INT

            type2 = parse_expr()
                            operator = parse_operator();                  // DIV
                            type1 = parse_expr()
                                        type = parse_primary()            // k
                                                  type = lookup_type("k") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // l
                                                  type = lookup_type("l") = INT;
                                                  return INT;
                                  = INT
                            type  =  type_check(operator, type1, type2)
                                  =  type_check(PLUS, INT, INT) = INT;
                            return INT;
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                            operator = parse_operator();                  // PLUS
                            type1 = parse_expr()
                                        type = parse_primary()            // i
                                                  type = lookup_type("i") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // j
                                                  type = lookup_type("j") = INT;
                                                  return INT;
                                  = INT
                            type  =  type_check(operator, type1, type2)
                                  =  type_check(PLUS, INT, INT) = INT;
                            return INT;
            type1  = INT

            type2 = parse_expr()
                            operator = parse_operator();                  // DIV
                            type1 = parse_expr()
                                        type = parse_primary()            // k
                                                  type = lookup_type("k") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // l
                                                  type = lookup_type("l") = INT;
                                                  return INT;
                                  = INT
                            type  =  type_check(operator, type1, type2)
                                  =  type_check(PLUS, INT, INT) = INT;
                            return INT;
            type2  = INT
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                            operator = parse_operator();                  // PLUS
                            type1 = parse_expr()
                                        type = parse_primary()            // i
                                                  type = lookup_type("i") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // j
                                                  type = lookup_type("j") = INT;
                                                  return INT;
                                  = INT
                            type  =  type_check(operator, type1, type2)
                                  =  type_check(PLUS, INT, INT) = INT;
                            return INT;
            type1  = INT

            type2 = parse_expr()
                            operator = parse_operator();                  // DIV
                            type1 = parse_expr()
                                        type = parse_primary()            // k
                                                  type = lookup_type("k") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // l
                                                  type = lookup_type("l") = INT;
                                                  return INT;
                                  = INT
                            type  =  type_check(operator, type1, type2)
                                  =  type_check(PLUS, INT, INT) = INT;
                            return INT;
            type2  = INT

            type = type_check(operator, type1, type2)
```

type checking for expressions: Example Execution

+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                            operator = parse_operator();                  // PLUS
                            type1 = parse_expr()
                                        type = parse_primary()            // i
                                                  type = lookup_type("i") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // j
                                                  type = lookup_type("j") = INT;
                                                  return INT;
                                  = INT
                            type  =  type_check(operator, type1, type2)
                                  =  type_check(PLUS, INT, INT) = INT;
                            return INT;
            type1  = INT

            type2 = parse_expr()
                            operator = parse_operator();                  // DIV
                            type1 = parse_expr()
                                        type = parse_primary()            // k
                                                  type = lookup_type("k") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // l
                                                  type = lookup_type("l") = INT;
                                                  return INT;
                                  = INT
                            type  =  type_check(operator, type1, type2)
                                  =  type_check(PLUS, INT, INT) = INT;
                            return INT;
            type2  = INT

            type = type_check(operator, type1, type2)
                 = type_check(PLUS, INT, REAL) = REAL
```

type checking for expressions: Example Execution

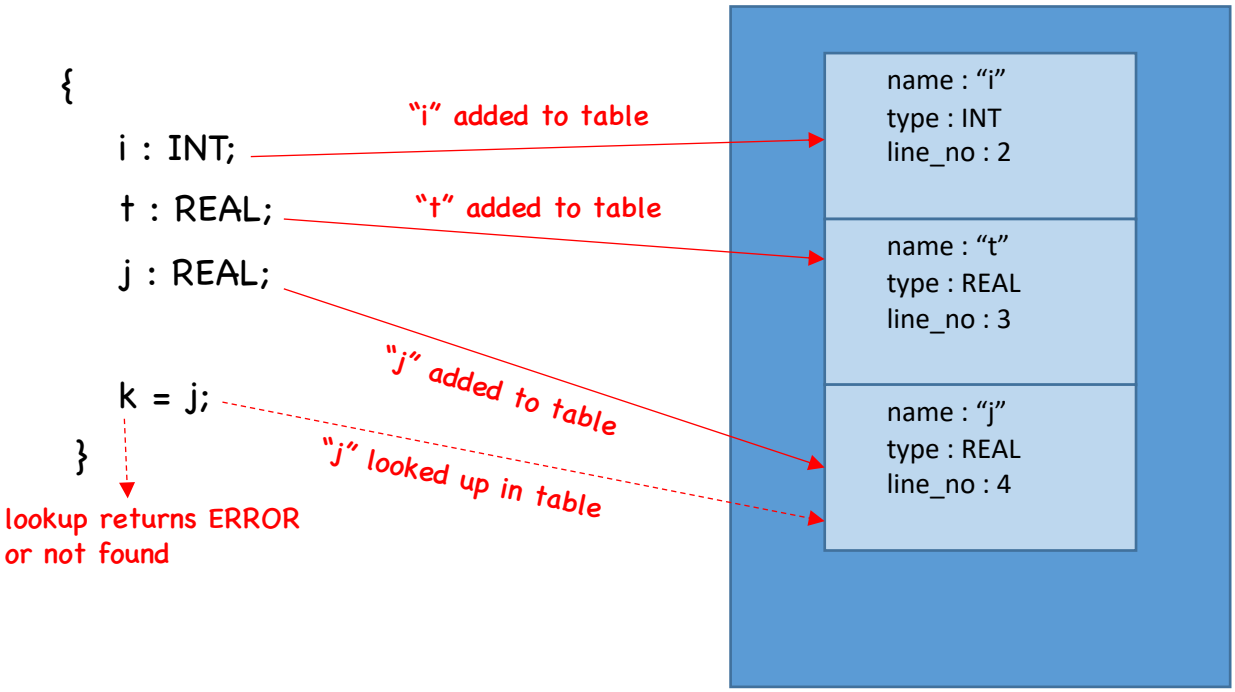
+ + i j / k l

```
parse_expr()
            operator = parse_operator();                                  // PLUS
            type1 = parse_expr()
                            operator = parse_operator();                  // PLUS
                            type1 = parse_expr()
                                        type = parse_primary()            // i
                                                  type = lookup_type("i") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // j
                                                  type = lookup_type("j") = INT;
                                                  return INT;
                                  = INT
                            type  =  type_check(operator, type1, type2)
                                  =  type_check(PLUS, INT, INT) = INT;
                            return INT;
            type1  = INT

            type2 = parse_expr()
                            operator = parse_operator();                  // DIV
                            type1 = parse_expr()
                                        type = parse_primary()            // k
                                                  type = lookup_type("k") = INT;
                                                  return INT;
                                  = INT
                            type2 = parse_expr()
                                        type = parse_primary()            // l
                                                  type = lookup_type("l") = INT;
                                                  return INT;
                                  = INT
                            type  =  type_check(operator, type1, type2)
                                  =  type_check(PLUS, INT, INT) = INT;
                            return INT;
            type2  = INT

            type = type_check(operator, type1, type2)
                 = type_check(PLUS, INT, REAL) = REAL

            return REAL
```

Symbol Table Entries

- The symbol table contains names and attributes of the names

- For your project 3, the local symbol table will consist of a vector of structures (or a map). Here is an example program and corresponding symbol table



- when introducing a name (adding a name to the table), you want to make sure that it does not conflict with another name already declared in the same scope
- when using a name, you want to make sure that the name was declared in the current or in some ancestor scope