

Chapter 6

Semantics

Recall :

Programming Language **syntax** means what the language constructs look like.

Programming Language **semantics** means what those language constructs actually do (meaning).

Programming language semantics are much more complex to express than the syntax. Programming language semantics can be specified by :

1. The Programming language reference manual (most common and simple).
2. Translator (Compiler or Interpreter).
By Experiment. Execute programs to find out what they do.
Machine dependent (generally it is not portable).
3. Formal Definition (mathematical model). It is complex and abstract.

We will mainly be using the first method.

We will also use ALGOL-like languages in our discussion

Binding

Using names or identifiers in a programming language is a basic, fundamental abstraction - variable names, constant names, procedure and function names are all examples of this.

Related to names is the concept of:

location. Simply put, the location is the address of the name in memory.

Value. Another thing related to the name is the **value**, which is the storable quantity in memory.

But how is the meaning of names determined?

Answer: It is determined by its **attributes** (properties associated with it).

for example :

```
const n = 15;
```

In this declaration, we associated 2 attributes :

1. It is a constant name.
2. it has a value of 15.

Another example :

```
var  
x:integer;
```

again, 2 attributes are associated with the name x :

1. It is a variable name.
2. It is of integer type.

Another example :

```
function compute (n:integer, x:Real):Real;  
Begin  
.  
.  
end;
```

Associated with the name *compute* (function name) is :

1. It's type : a function name.
2. Number and type of parameters : it takes 2 parameters, one of type integer .
and one of type Real .
3. It's return value : The function returns Real .
4. The code body of the function.

Another example :

var

$y : ^{\wedge} integer; \equiv int *y;$

Associated with the name y is :

1. It's a **variable** name.
2. It is of a **pointer** to an **integer** type.

Notice that in all the examples above, all attributes are determined at declaration.

However, we can **assign attributes** outside the declaration.

For example :

$x := 2;$

this means that we add a **new attribute** to the name x , which is the **value**.

In the example,

var

$y : ^{\wedge} integer;$

if we say,

$New(y);$

Then, in this case, we add a third attribute to the variable y which is the **location**.

When we first declared y , it pointed to junk (something random). When we used $new(y)$, pascal reserved a place in the memory the size of an integer and changed the reference to it (without having to name it, unlike C).

The process of associating attributes to names is called **Binding**. This happens at **Binding Time**.

Binding Time : The time during the translation(compile) process when the attribute is computed and associated to the name.

There are 2 kinds of binding times.

1. **Static Binding** : binding which occurs before execution. We call those attributes **static attributes**.
2. **Dynamic Binding** : binding which occurs during execution.

Examples :

1. `const n = 15;`

is a **static attribute**. This is because the attributes constant name and value=15 are assigned during compilation.

2. `x:integer`

The attributes variable name & integer type are also **static attributes**.

However, when we say `x:=2`, the attribute value=15 is a **dynamic attribute** because it is assigned during execution.

3. `y^:integer ;`

the attributes variable name & pointer to integer type are **static attributes**, while `new(y)`, the added attribute location is a **dynamic attribute**.

Binding can be performed prior to translation.

As an examples in Pascal:

- Binding reserved words(names), Data structure types, and Array storage layout are predetermined at **language definition time**.
- Binding the values for the **integer** type (Range of integers) and the values for the **Boolean** type (true, false), The constant maxint in Pascal are defined at implementation time.

↳ a constant value that's equal to maximum integer on the machine.
↳ it's value is determined during implementation time.

In short, Binding can be performed at :

- Language definition time.
- Language implementation time.
- Translation time. "**most attributes are defined here**"
 - at lexical analysis.
 - at syntax analysis.
 - at code generation.

All the above bindings are static.

- Execution time. This binding is dynamic.

Symbol Table

The Symbol Table is a special data structure used to maintain the binding during the translation process

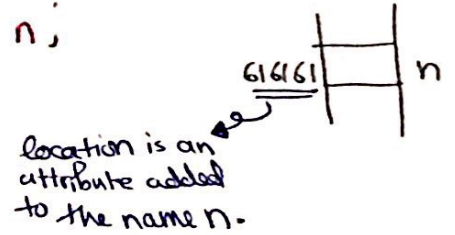
Environment

The Environment is the memory allocation part of the execution process. ie, binding names to the storage locations is called Environment.

Memory

Memory is the binding the storage locations to values.

`int n;`



Declarations and Blocks

Declarations are the principle method to establish binding. There are 2 types of declarations:

1- Explicit Declaration:

Pascal:

Var

X:integer;

Ok:Boolean;

Algol68:

Begin

Integer x;

Boolean ok;

End

Ada:

Declare

X:integer;

Ok:Boolean;

C:

Int x;

2- Implicit Declaration:

⚠ In (ie) PASCAL there's no implicit declaration.

The variable is declared when it is used.

for example, in C: `int n = 10`

Declarations are associated with **blocks**. There are 2 types of blocks:

- 1- Main Program Block.
- 2- Procedure or function block.

for Example in Pascal:

```
Program Test;  
Var  
.  
.  
.  
procedure P;  
  Var  
  .  
  .  
  .  
  Begin  
  .  
  .  
  .  
  End;  
.  
.  
function q:integer;  
  Var  
  .  
  .  
  .  
  Begin  
  .  
  .  
  .  
  End;  
Begin(*main*)  
.  
.  
End.
```

block

block

main program BLOCK!

In Algol:

```
Begin  
  Integer X;  
  Boolean Y;  
  .  
  X := 2;  
  Y := True  
  .  
End
```

declaration included within block.

In Ada:

```
Declare  
  X : Integer;  
  Y : Boolean;  
Begin  
  X := 2;  
  Y := 0;  
End;
```

Declarations bind different attributes to names especially the static type of attributes.
Note that the declaration itself has an attribute, which is the position of the declaration in the program. This is important to determine the **scope/visibility** of the variable.

scope of Declaration

The scope of declaration is :

The region of the program over which the declaration covers. In block structured languages, such as PASCAL, the scope of declaration is limited to the block in which is declared/appears and all other nested blocks. Contained within this block.

In fact, a language like PASCAL has the following scope rule :

The scope of declaration extends from the point it is declared to the end of the block.

for example :

```
Program scope;
  VAR X : Integer;

  Procedure P;
    VAR X:Real;
  BEGIN
    .
  END;

  Procedure q;
    VAR Z:Boolean;
  BEGIN
    .
  End;

  BEGIN(*main*)
    .
  END.
```

The diagram illustrates the scope of variables in the code. A red arrow labeled "scope of X" points from the first declaration of X to the end of the program. A red arrow labeled "scope of Z" points from the declaration of Z to the end of procedure q. A red arrow labeled "scope of global X" points from the first declaration of X to the end of the program.

In Algol 60:

```
A:BEGIN
  Integer X;
  Boolean Y;
  X:=2;
  .
  .
B:BEGIN
  Integer c,d;
  .
  .
End;
.
.
End
```

x, y have scope both in blocks A & B, while c, d have scope in block B only.

In Modula-2:

```
Module Ex;  
  Procedure P;  
  begin  
    x:=2;  
  end P;  
  
var  
  x:integer;  
  
begin  
  .  
  .  
End Ex.
```

Implicit declaration.



In Modula-2 the declaration extends all over the block backward & forward not just from the point of declaration.
The scope of x extend all over the program block.

Important Note:

In block structured languages such as Pascal is that:

The declarations in nested blocks takes precedence over previous declarations.

Ex:

```
Program ex;  
Var x:integer;  
Procedure P;  
  Var x:Real; (*x local to P *)  
  Begin  
    X:=3.5;  
  End;  
Begin  
  X :=2; (* x is the global *)  
End.
```

visibility of global X.



That is, the global x can't be accessed inside P, we say the "global X" has a scope hole inside P.

That is why we differentiate between scope and visibility.

Visibility: The area where the name apply(excluding holes).

scope: Including holes.

Symbol Table

All the declarations and binding are established by a structure called the symbol table. In addition, the symbol table must maintain the scope of declaration. Different data structures can be used in the symbol table :

1. Hash Table --> static.
2. Linked List --> dynamic.
3. Tree Structure --> dynamic.

To maintain the scope of declarations correctly, the declarations should be processed using the stack concept(FILO). When entering a block, declarations are processed and attributes are added/bound to the symbol table (pushed to stack). When exiting from the block, the binding (the attributes) provided in the block are removed/popped from the stack.

Think of the symbol table as a set of names, each of which has a stack of declarations associated with it. The top of the stack is the current active declaration.

For example, consider the following pascal program :

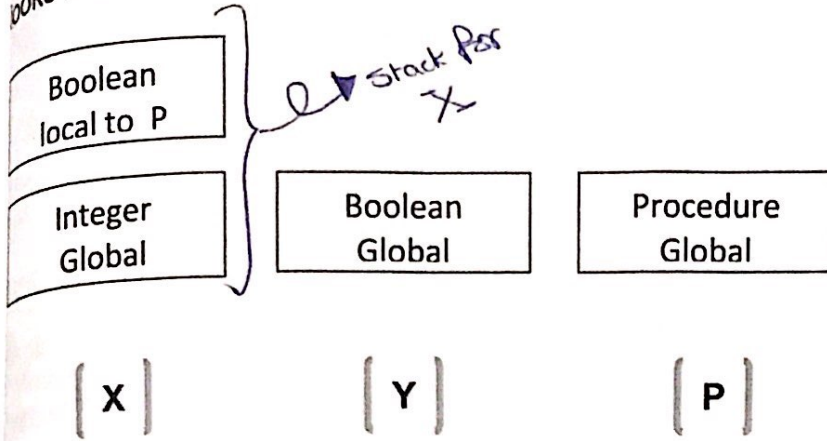
```
Program symbol_table;
Var X:integer;
    Y:boolean;
Procedure P;
  Var x:boolean;
  Procedure Q;
    Var y:integer;
    Begin
      .
    End; (*Q*)
  Begin
    .
  End; (*P*)
Begin (*main*)
.
End.
```

void function in C
it doesn't return
value.

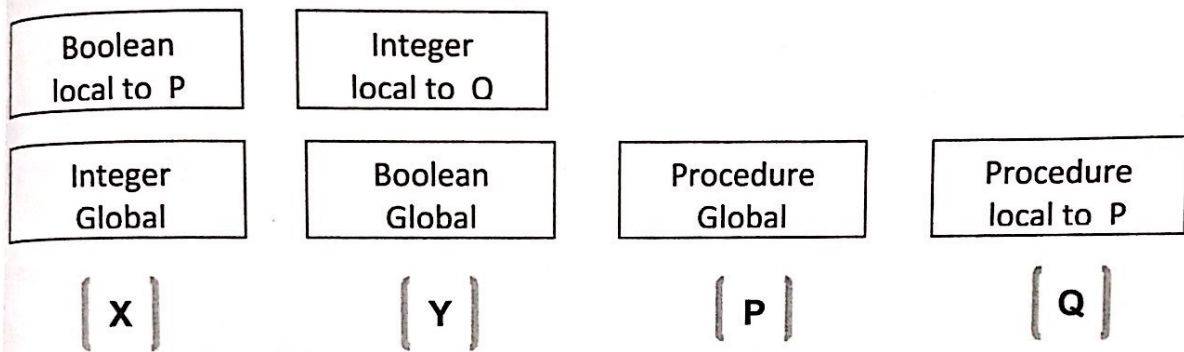
Global local to P,
main

There are 4 names declared in the program, X, Y, P, Q

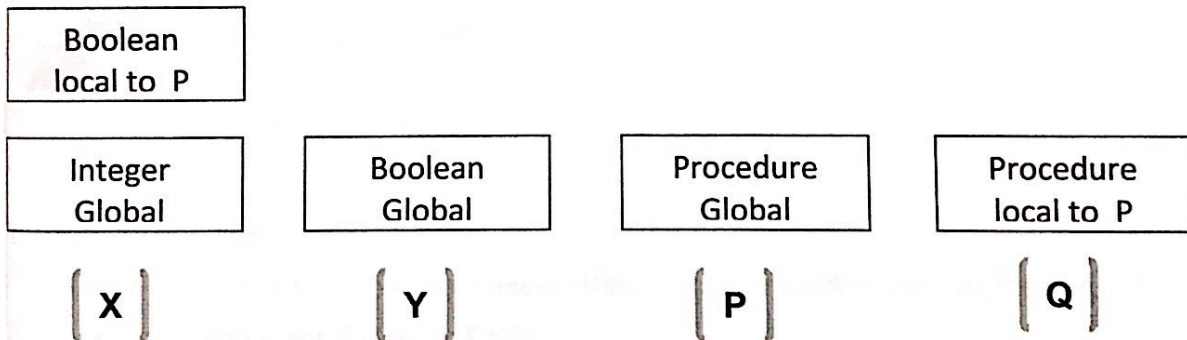
After Processing the global variables X, Y and procedure P, the symbol table looks like:



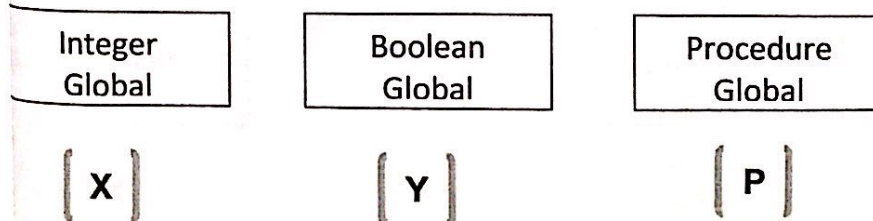
After processing procedure Q inside P ,



After exiting from the body of procedure Q



After exiting from the body of procedure P



This scheme of scoping is called **Lexical scoping** or **Static Scoping**.

There are two types of scoping:

- 1- Lexical (static) scoping. deals with the symbol table according to declarations. "built on scanning the source code top-down"
- 2- Dynamic scoping. "Symbol table is built when executing the code"

Example:

```
Program Scoping;  
Var X:integer;
```

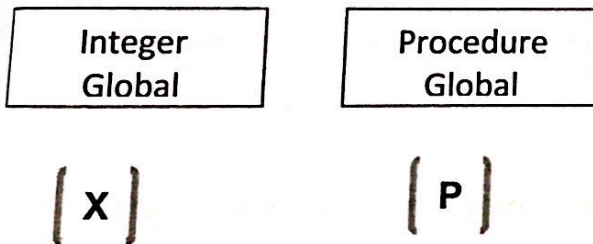
```
Procedure P;
```

```
Begin  
Write(X);  
End; (*P*)
```

```
Procedure Q;  
Var X:integer;  
Begin  
X:=2;  
P;  
End; (*Q*)
```

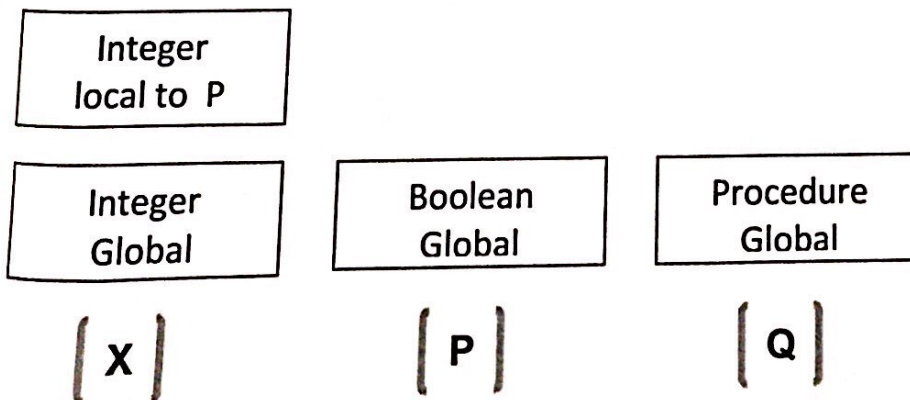
```
Begin (*main*)  
X:=1;  
Q;  
End.
```

- Now using the Lexical Scoping, the symbol table looks like:



The value 1 is printed.

- Using Dynamic Scoping, the symbol table processes declarations as they are encountered in the EXECUTION.



The value 2 is printed. Most block structured languages perform Lexical scoping. LISP Dynamic scoping.

Allocation and Environment

↳ the allocation of memory locations to names.

- Symbol table maintains in the declaration the binding of attributes to names.
- Environment is binding names (or associating names) to locations.
- Environment may be constructed :
 1. **Statically** (at load time) - Fortran.
 2. **Dynamically** (at execution time) - Lisp.
 3. **Mixture** (block structured languages such as Pascal, C, Modula-2, Ada, ...)

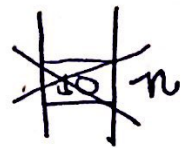
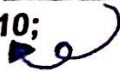
Some allocations are performed statically and some dynamically.

Global variables: statically.

Local variables: dynamically.

Some names are not bound to locations at all, for example:

const n=10;



The compiler replaces all occurrences of n by 10 in the block during execution with no need to allocate space for n.

- Environment in block structured languages binds locations to **local variables** in a **stack-based** fashion.
- During execution, on entering each block, the variables declared at the beginning of the block are allocated. On exit from that block, the same variables are deallocated.
-

Example:

```
Program Test;  
Var x, y : integer;
```

```
Procedure A(x:integer);  
Var y, z : real;
```

```
Begin
```

```
End; (*A*)
```

```
Procedure B(n:boolean);  
Var y, z : real;
```

```
Procedure C(h,p:real);  
Var x,y : integer;
```

```
Begin
```

```
End; (*C*)
```

```
Begin
```

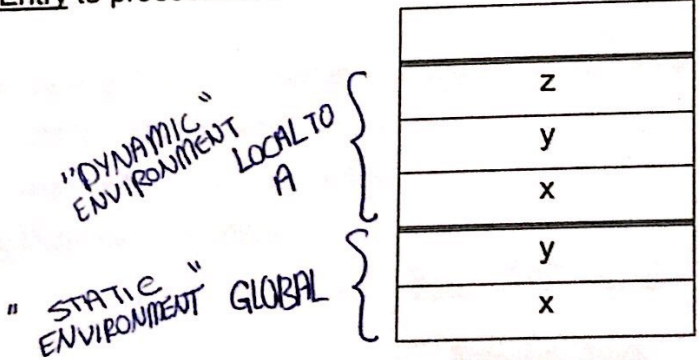
```
End; (*B*)
```

```
Begin (*main*)
```

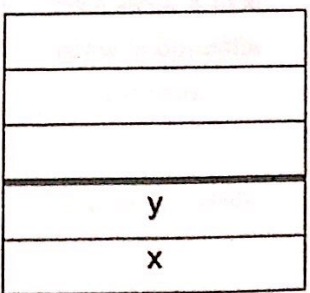
```
End.
```

Then the stack will look like:

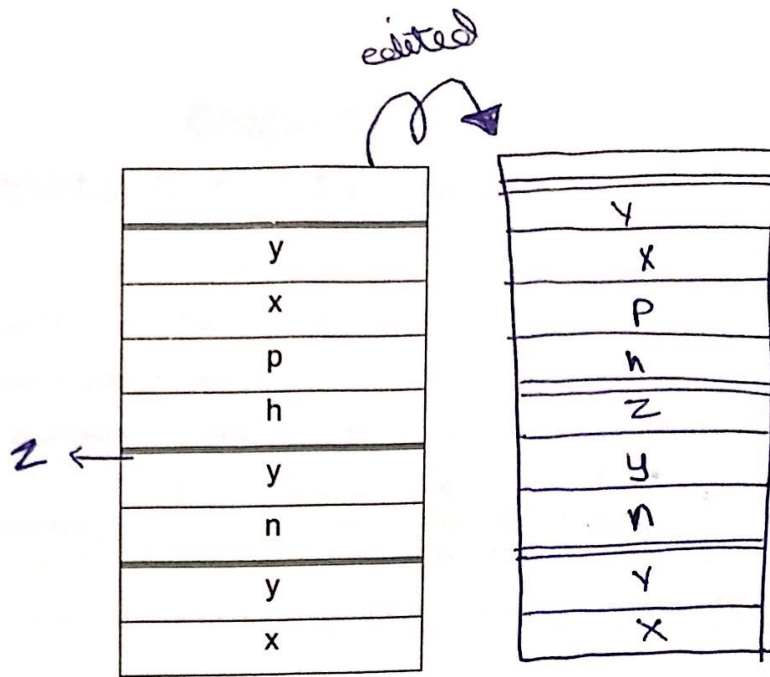
After Entry to procedure A:



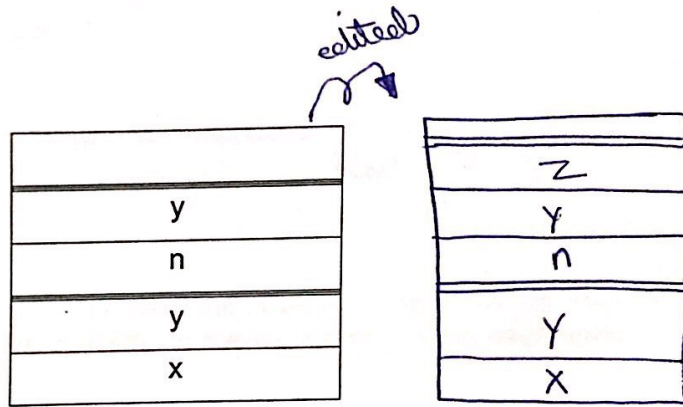
After Exit from A:



After Entry B & C:

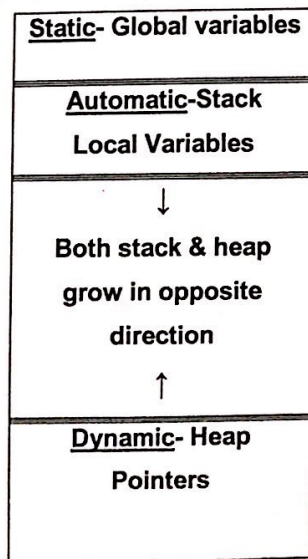


Exit from C:



Generally, in block structured languages, there are 3 kinds of allocation in the Environment:

- 1) Static – Global variables.
- 2) Automatic – Local variables.
- 3) Dynamic – Pointers.



* parameters & variable of procedures & function calls are pushed to the stack.

Chapter 7

Syntax Directed Translation

Lexical Structure \rightarrow Systematic algorithms exist \rightarrow Finite State Automata.

Syntax Structure \rightarrow Systematic algorithms exist \rightarrow Push Down Automata.

Semantic Structure \rightarrow Unfortunately, no systematic algorithm.

However, there is a framework for **intermediate code generation**, which is an extension of the context-free grammar called **syntax-directed translation**.

Semantic Analysis

In syntax-directed translation, the algorithm allows what is called a **semantic action**, which is simply a **subroutine (procedure/function) attached to some of the production rules** of the context-free grammar.

A semantic action or a semantic rule is simply an output action added (associated) to the production rule of the grammar. For example, given the production

$A \rightarrow a$

the semantic action is simply

$A \rightarrow a \# B$

to recognize semantic action

where **B is the semantic action**.

*α, B are functions.
Function $\alpha() \{ \}$ Function $B() \{ \}$*

Take the production

$A \rightarrow XYZ \# \alpha$

assume that α is the semantic action/rule, then in the syntax-directed translation scheme, the semantic action α is called/executed whenever the parser recognizes or accepts a sentence w derived from A in **top-down parsing**.

$A \rightarrow XYZ \rightarrow w \in L(G)$

In **bottom-up parsers**, the semantic action α is called whenever **XYZ is reduced**.

Generally, compilers generate/translate source code into another format which is easier for the compiler to understand (evaluate). This code is called **intermediate code**. There are different kinds of intermediate code:

1. Postfix Code: for example,

$A * B * (C * D) * E$

becomes

$A * B * C * D * E$

in postfix.

Another example:

```
if a
  x
else
  y
```

would become

... a x y ? ...

in postfix.

Or another example:

...

! We can't change the order of operands.
 $BC - B * X$

```

    if a
    if (c - D)
        a + c
    else
        a * c
    else
        a + b
    ...
    becomes
    ... a c D - a c + a c * ? a b + ? ...
    in postfix.

```

2. Three Address Code (TAC) : Each instruction has at most 3 components.

for example : \downarrow

$$\frac{-w * x + (y + z)}{2}$$

would be

- (1) -, w *temporary location*
- (2) *, (1), x *Unary operation.*
- (3) +, y, z *Binary operation.*
- (4) +, (2), (3)

in TAC.

Another Example :

```

if (x > y)
    z = x
else
    z = y + 1

```

would be

- (1) -, x, y
 - (2) JGZ, (1), (6)
 - (3) +, y, 1
 - (4) =, z, (3)
 - (5) JMP, (7)
 - (6) =, z, x
 - (7)
- if statement completed.*

in TAC.

3. Quadruples : Another form of intermediate code, which has at most 4 components. for example : $-w * x + (y + z)$
would be :

operation	operand 1	operand 2	result
-	w	-	R1
*	R1	x	R2
+	y	z	R3
+	R2	R3	R4

Table of Quadruples