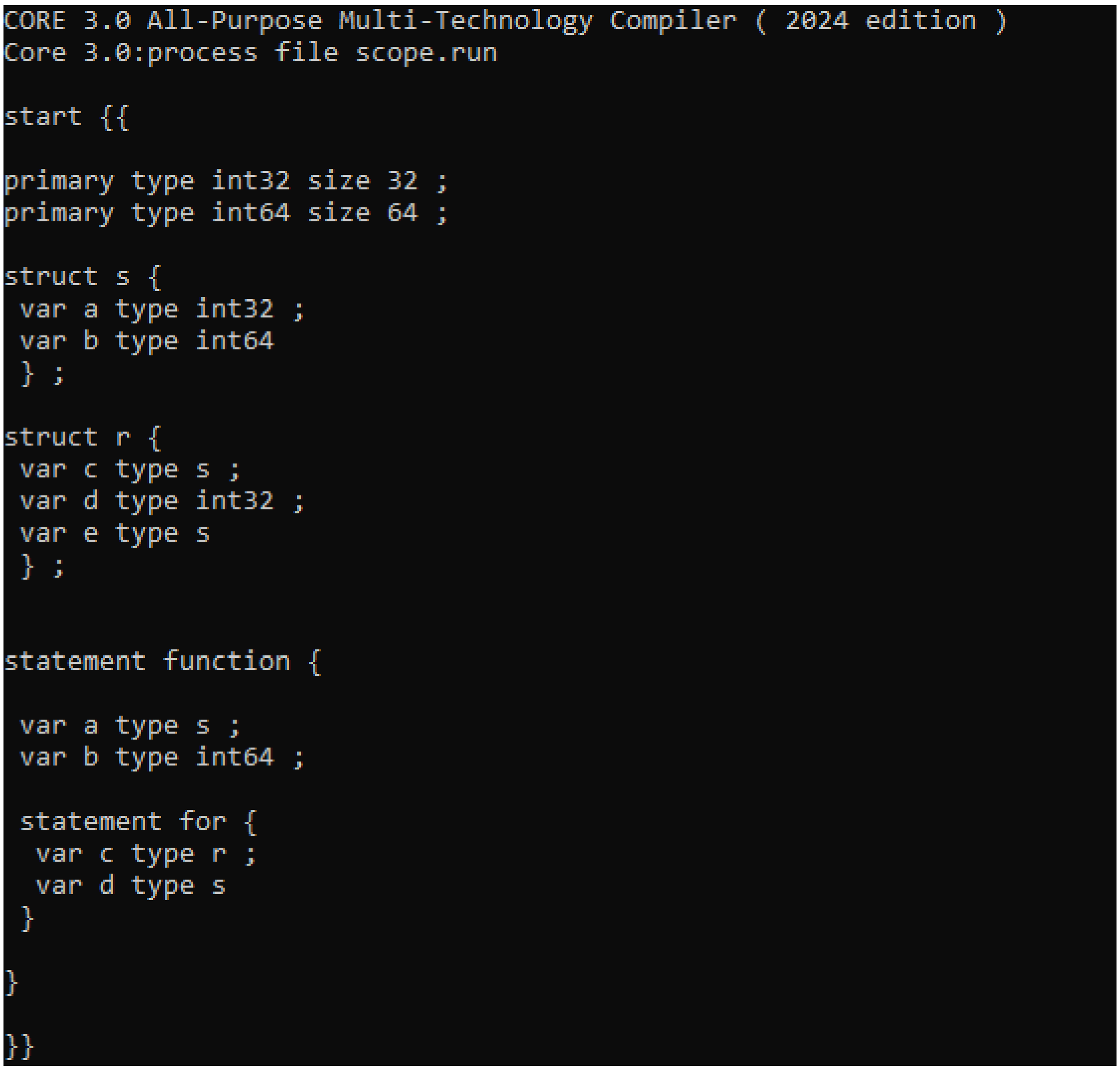
Core 3.0 2024 Edition
Implementation of the “scope” example including the definition of primitive types and structures

We begin the discussion of the topic by defining the types of statements that the language provides.

```
USER LANGUAGE

BEGIN
primary&: primary|(str) type|.type|(frw) size|.size|(frw)
var&: var|.var|(frw) type|.type|(frw)
struct&: struct|.name|(frw) {|.%statements|(**) ;|(**)|(opt) }
statement&: statement|(frw) {|.%statements|(**) ;|(**)|(opt) }
END
```

Let's immediately see the type of source in the example:



```
CORE 3.0 All-Purpose Multi-Technology Compiler ( 2024 edition )
Core 3.0:process file scope.run

start {{

primary type int32 size 32 ;
primary type int64 size 64 ;

struct s {
 var a type int32 ;
 var b type int64
 } ;

struct r {
 var c type s ;
 var d type int32 ;
 var e type s
 } ;


statement function {

 var a type s ;
 var b type int64 ;

 statement for {
  var c type r ;
  var d type s
 }

}

}}
```

Let's also immediately define a pre-processing map that we will need shortly.

```
PREPROCESSING MAP

BEGIN
#redef 1
#shadow 2
END
```

We begin the implementation by defining an object that contains the name and reference to the types that are defined which can be primary or secondary such as structures (struct).

```
public function new::types ( null ) {{
        set .t = new types ;
        return .t

}} ;

set ..types = exec new::types ( null )
```

An instance of this object is set as "global". The object has two functions: Addition of a type with associated reference, checking that there are no redefinitions:

```
public function types.add_type ( :name , :ref ) {{

        for ( set .i = 0 ; .i < size ::name ; .i = .i + 1 ) {{

                if ( ::name[ .i ] == :name ) {{ rule error "Redefinition of " + :name }}

        }} ;

        set[] ::name = :name ;
        set[] ::ref = :ref

}} ;
```

And return a reference starting from the type name, checking that it is a previously defined type:

```
public function types.get_type ( :name ) {{

for ( set .i = 0 ; .i < size ::name ; .i = .i + 1 ) {{

                if ( ::name[ .i ] == :name ) {{ return ::ref[ .i ] }}

        }} ;

        rule error :name + " is not a defined type."
}}
```

Otherwise, in both cases an error will be generated that blocks the compilation.

An "instance" type object is defined, which contains various information, in particular the type of instantiations for each variable, with the size, and in some cases with the scope and state. So the constructor this time will have four parameters:

```
public function new::instance ( :name , :size , :offset , :state ) {{

        set .i = new instance ;

        set .i.name = ( :name ) ;
        set .i.size = ( :size ) ;
        set .i.offset = ( :offset ) ;
        set .i.state = ( :state ) ;

        return .i
}} ;
```

Let's start implementing “primary type”, which defines a primitive type, and adds a null variable instance of the size expressed in size.

```
public function primary.autoinit ( null ) {{

        exec ..types.add ( text ::type , ::__SELF ) ;
        set[] ::instaces = exec new::instance ( "" , int ::size )

}}
```

Now let's move on to implementing the "scope" type object, which contains variables and instances.

```
public function new::scope ( null ) {{

        set .s = new scope ;
        set .s.offset = 0 ;
        return .s

}}
```

The copy function of "scope", providing for the offset assignment, is the creation of a reference to the higher level scope to check whether instances of variables are not created that shadow the variables of the higher level statements in the statement tree . The offset value is also initialized with the higher level scope offset value.

```
public function copy::scope ( :s ) {{

        set .s = new scope ;
        set .s.offset = ( :s.offset ) ;
        set .s.super = :s ;
        return .s

}} ;
```

The check serves to verify the possible redefinition error or the creation of a shadow variable (the latter case, however, has been partially implemented in this example and is currently not working).

```
public function verify ( :scope , :name , :type ) {{

        set .r = 0 ;

        if ( :type == #shadow ) {{
                if ( :scope.super ) {{ .r = exec verify ( :scope.super , ( :name ) , #shadow )  }}
        }} ;

        for ( set .i = 0 ; .i < size :scope.variables ; .i = .i + 1 ) {{
                if ( :scope.variables[ .i ] == :name ) {{ return :type }}
        }} ;

        return ( .r )
}} ;
```

Let's start the implementation for struct by saying that each struct has its own scope object:

```
public function struct.autoinit ( null ) {{

        set ::scope = exec new::scope ( null )

}}
```

The function of adding variables to struct provides for the verification of non-redefinition and the addition to the scope of the variable:

```
public function struct.add_variable ( :name , :ref ) {{

        if ( exec verify ( ::scope , :name , #redef ) == #redef ) {{ rule error "Redefinition of " + :name }} ;

        set[] ::scope.variables = ( :name ) ;

        exec add_variable ( ::scope , "." + :name , :ref , 0 )

}} ;
```

The add variable function (add_variable) adds all instances of the variable to the scope, and the offset for each variable is also calculated:

```
public function add_variable ( :scope , :name , :ref , :state ) {{

        for ( set .i = 0 ; .i < size :ref.instances ; .i = .i + 1 ) {{

                set[] ::scope.instances = exec new::instance ( :name + :ref.instances[ .i ].name
                , ( :ref.instances[ .i ].size )
                , ( :scope.offset )
                , :state ) ;

                :scope.offset = :scope.offset + :ref.instances[ .i ].size

        }}

}}
```

Let's move on to the implementation of the statements. Each statement has its own scope object, which is "new" if the statements are at the root, or a copy if the statements are nested.

```
public function statement.autoinit ( null ) {{

       if ( ::__CALLER.scope ) {{ set ::scope = exec copy::scope ( ::__CALLER.scope ) }} else {{ set ::scope = exec new::scope ( null ) }}

}}
```

The add variable function for statements is similar to that for structs, as you can see:

```
public function statement.add_variable ( :name , :ref ) {{

        if ( exec verify ( ::scope , :name , #redef ) == #redef ) {{ rule error "Redefinition of " + :name }} ;

        set[] ::scope.variables = ( :name ) ;

        exec add_variable ( ::scope , "" + :name  , :ref , 0 )

}} ;
```

As regards the "var" statement, which allows the definition of variables, the code is as follows:

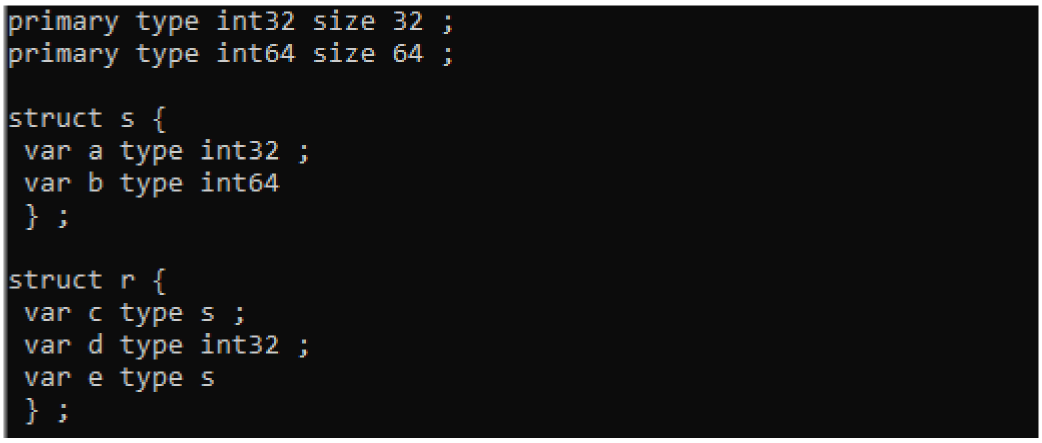
```
public function var.autoinit ( null ) {{

        exec ::__CALLER.add_variable ( text ::var , exec ..types.get_type ( text ::type ) )

}} ;
```

Then the variable is added either to struct or to statement, taking into account its type, which must be defined.

And then there is all the printing work, valuable for verification and for this example:

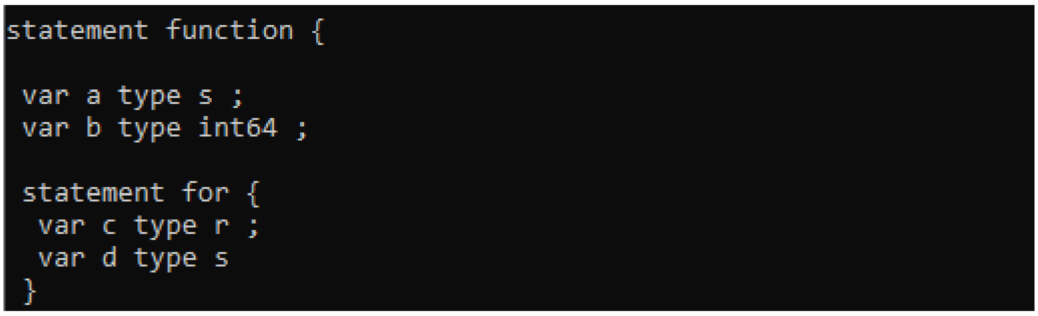
```
public function instance.print ( null ) {{
        print ( text ::name ) + "\t" + ( text ::size ) + "\t" + ( text ::offset ) + "\t" + ( text ::state ) + "\n"
}} ;

public function struct.autoexec ( null ) {{
        print "struct " + ( text ::name ) + "{\+\n" ;
        exec ::scope.print ( null ) ;
        print "\-\n}\n"
}} ;

public function statement.autoexec ( null ) {{
        print "statement " + ( text ::name ) + "{\+\n" ;
        exec ::scope.print ( null ) ;
        print "\-\n}\n"
}} ;

public function scope.print ( null ) {{

        for ( set .i = 0 ; .i < size ::instances ; .i = .i + 1 ) {{ exec ::instances[ .i ].print ( null ) }}

}}
```

Starting from the source that we have seen and that we show:



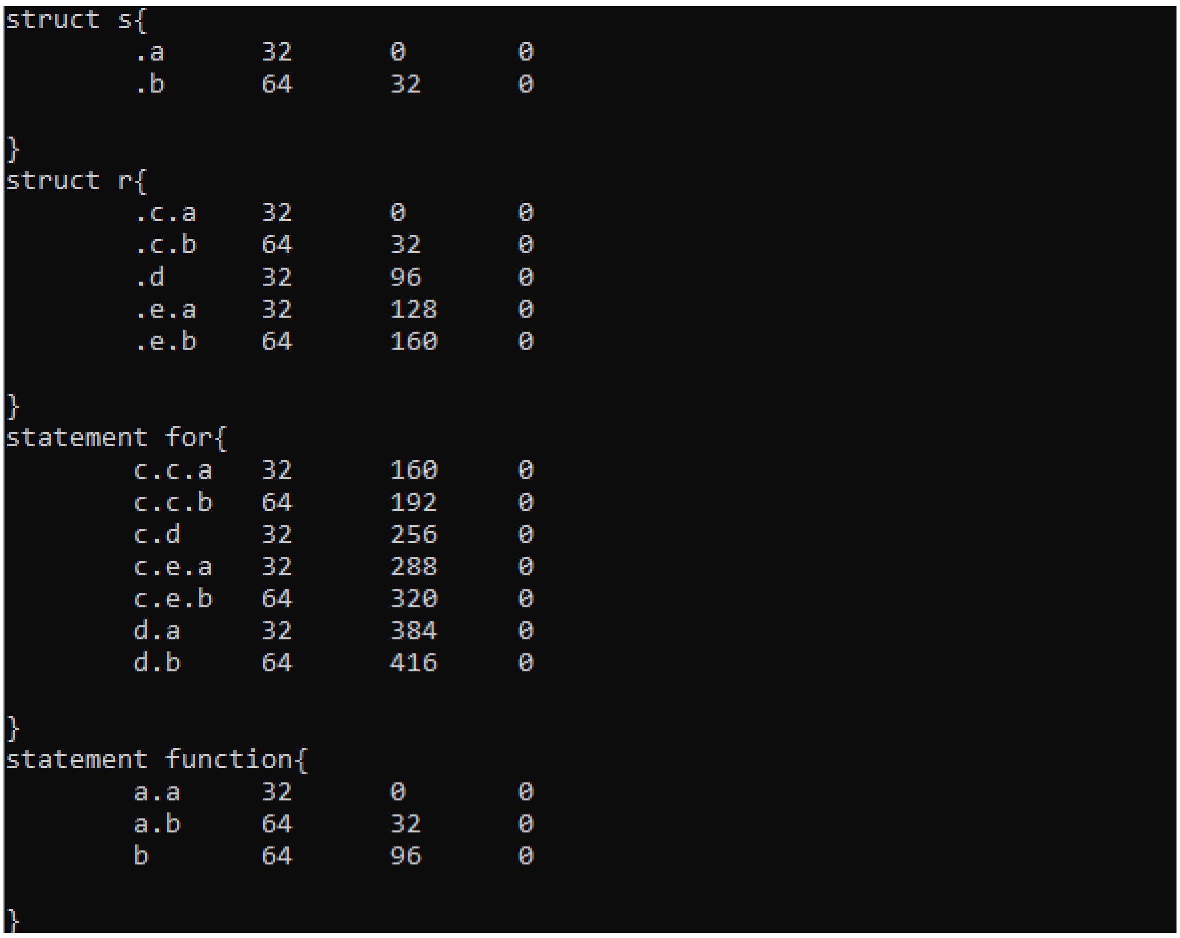
```
primary type int32 size 32 ;
primary type int64 size 64 ;

struct s {
 var a type int32 ;
 var b type int64
 } ;

struct r {
 var c type s ;
 var d type int32 ;
 var e type s
 } ;
```

Where two primitive types and two structures are defined. From a possible source like the following:



```
statement function {

 var a type s ;
 var b type int64 ;

 statement for {
  var c type r ;
  var d type s
 }
```

We get the following output, where for each structure and for each statement, we have variable name, size, offset, and redefinition.



```
struct s{
        .a      32      0       0
        .b      64      32      0

}
struct r{
        .c.a    32      0       0
        .c.b    64      32      0
        .d      32      96      0
        .e.a    32      128     0
        .e.b    64      160     0

}
statement for{
        c.c.a   32      160     0
        c.c.b   64      192     0
        c.d     32      256     0
        c.e.a   32      288     0
        c.e.b   64      320     0
        d.a     32      384     0
        d.b     64      416     0

}
statement function{
        a.a     32      0       0
        a.b     64      32      0
        b       64      96      0

}
```

This is an implementation that shows how it is possible to create high-level languages, compiled at low level, at the ASSEMBLER level, through an all-purpose compiler.

Thank you