Team Fourtran Final Project V2

**Introduction:**

Introducing 64tran, a new object-oriented language that also fully supports functional programming. 64tran currently supports 4 types: strings, bools, reals, and integers. Since 64tran is an object-based language, each type has a list of routines. The selling point of our language is that routines are objects just like any integer, or real, or string, thus they can be used like any other object.

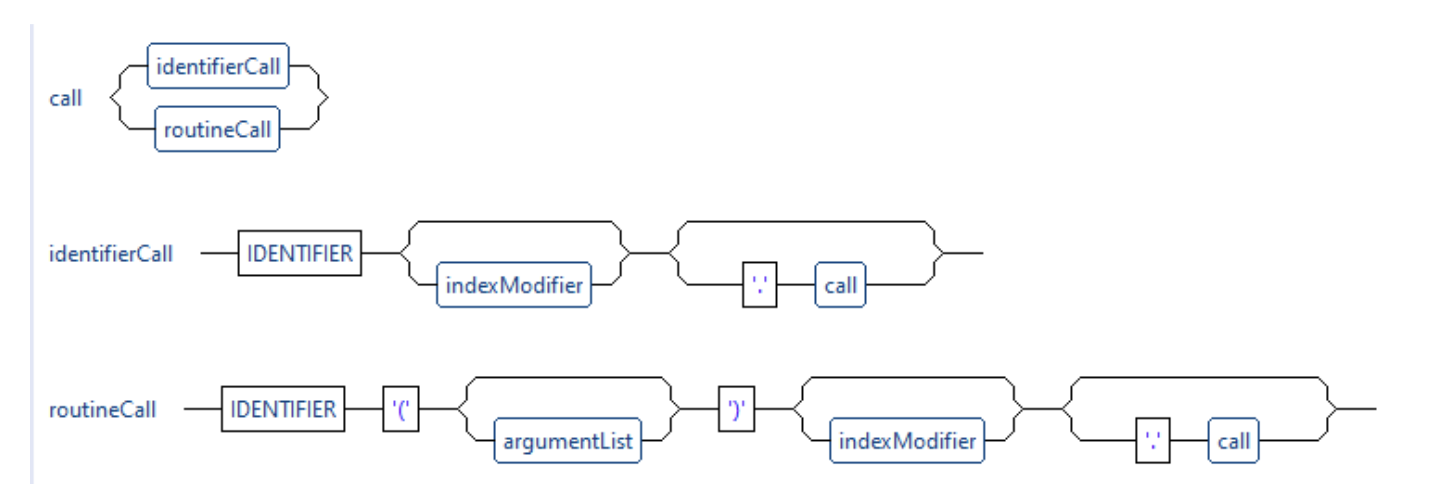
**64tran Grammar:**

64tran’s grammar takes inspiration from python. There is no program header or main class. To print “hello world”, all you need is *print(“Hello World!”)*! There is no semicolon either; all statements are ended with a newline. We couldn’t implement indent-based scopes, so all scopes end with an *end [name]* statement, where *[name]* is either the name of the type, the name of the routine, *if*, *else if,* *else*, or *while*.

Strings can be created using single quotes or double quotes, and booleans are either *true* or *false.* The main body of a program is comprised of statements, such as assignment statements, variable declarations, and routine or field calls. We have included all railroad diagrams in a different file. The most interesting part of the grammar are the types and the calls.

**64tran calls:**

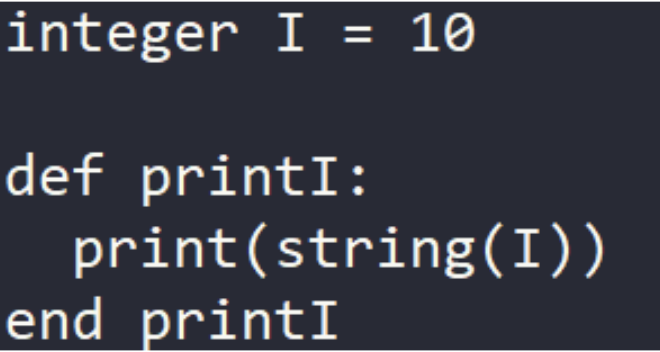
Aside from types, the only other interesting part about the 64tran grammar are its terms and calls, as seen in the next figure.



What makes calls so interesting is that they are recursive. Like in Java or C++, one can identifier calls together.

**64tran Scopes:**

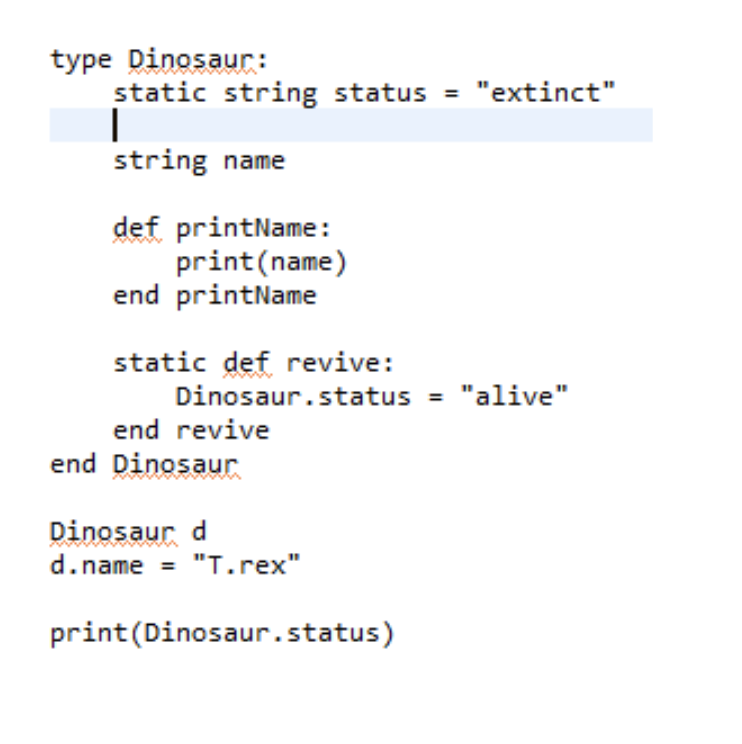
64tran uses a unique scoping system. Each type has its own scope, as does each routine, and if, else if, else, and while all have a separate scope. All inner scopes have access to variables and types declared in outer scopes. Variables in an inner scope can have the same name as a variable form an outer scope, but outer scopes cannot access a variable declared in an inner scope. The following figure demonstrates this:



This figure also demonstrates the casting ability capabilities of each of the predefined types. The *print* routine only accepts a *string* input, but *integers*, *reals*, and *bools* can be cast to a *string* using the above syntax. This is also referred to as the *static operator\_parenthesis*, because the *operator\_parenthesis* method of the *string* type is being statically called.

**64tran Types:**

One can declare custom types in 64tran with fields, static fields, routines, and static routines. All variable declarations must come before the routine declarations. Normal routines can access the type’s field without any modifiers, but it must make a static call to access the type’s static fields. The following program shows how to define a type and access it’s fields.



**Routines are objects:**

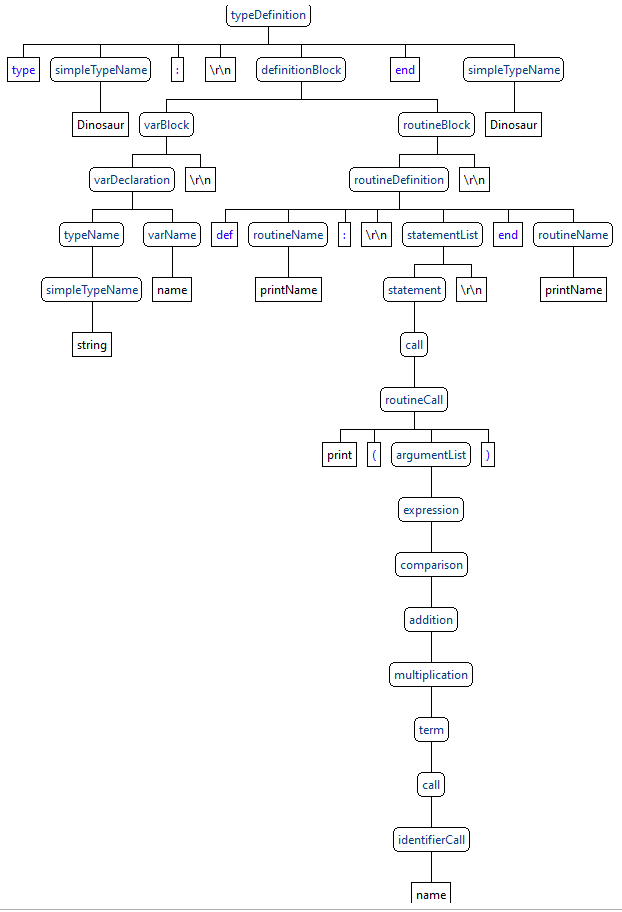
The big selling point for 64tran is that *routines are objects*. You can create variables that store routines. You can pass routines as parameters. You can create a routine that returns a routine. A routine variable or parameter is defined as such:

*routine of type1, type2 returns type3*

If there are no parameters, then *of type1, type2* is excluded. If there is no return type, *return type3* is excluded. So a routine variable that has no parameters and returns nothing is simply defined as *routine*.

**Parse Tree Diagrams:**

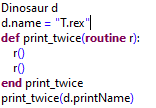
This is the parse tree diagram for the previously mentioned Dinosaur type without the static field/routine:



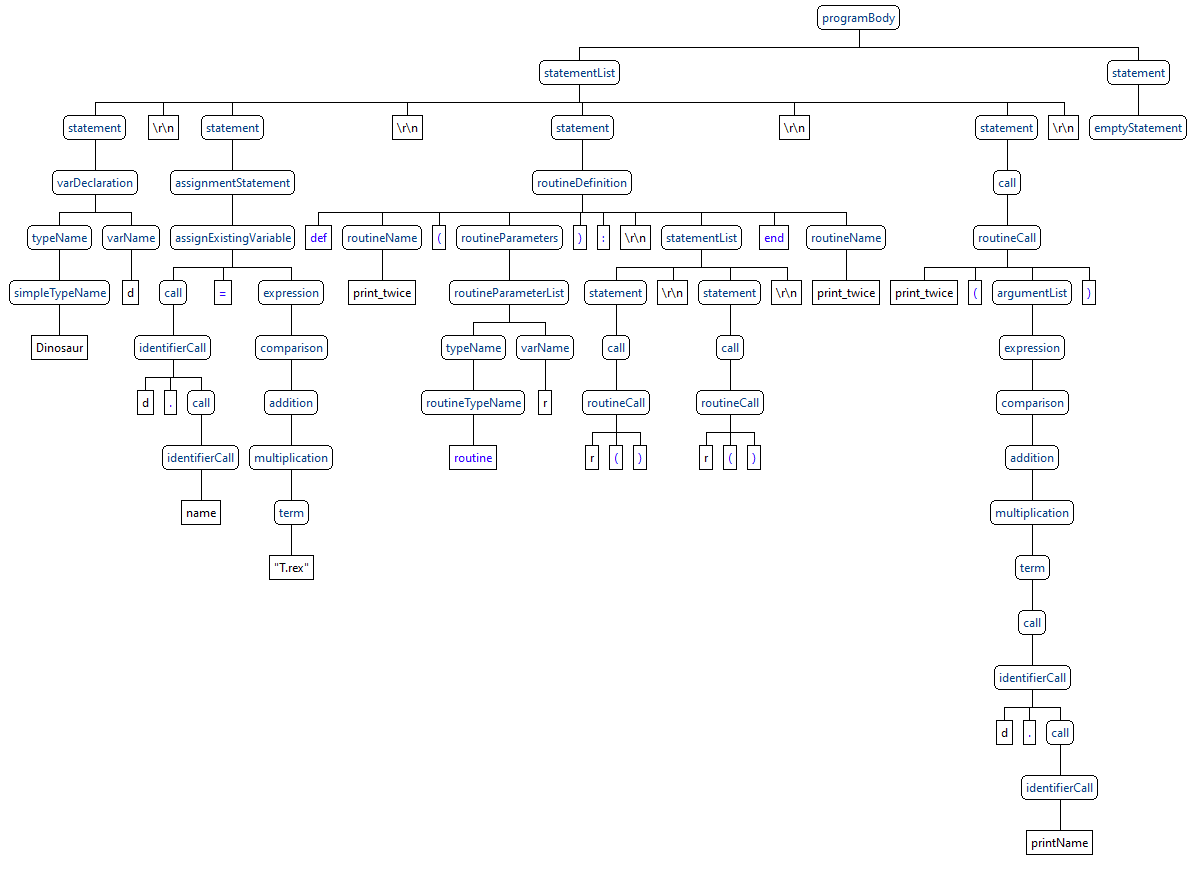
There are a few things of note:

1. There are two simpleTypeName contexts in this routine. The first one defines the name of the type and the second one declares the end of the type’s scope. The second simpleTypeName must be the same as the first, otherwise there will be a semantic error stating that the type was not closed properly.
2. The varBlock comes before the routineBlock. We decided that all variables must be declared before the routines. This is because, in Python, it variables should typically be declared before they are used in a routine, so we use wanted to enforce this policy.
3. The parse tree for expressions is very long. This is because we have several levels of operator precedence.

The next parse tree diagram demonstrates what we can do with a type. The code used to create is similar to the program seen previously, except now there is an additional routine called *do\_twice­*. The code is as follows:



It creates the following parse tree:



Much like types, routines must be ended with an *end [routine\_name]* or else there will be an error, although this error is thrown during the semantics pass, so it is syntactically correct. This program also demonstrates the recursive nature of the *call* grammar rule, which was by far the hardest rule to write the syntax and semantics for.

**Running the program:**

To run the program, run the main method found in *SixtyFortran.java*. It requires 1 argument, and there is an optional argument. The first argument is a path to your program, which must end with the *.f* extension. The second argument is the *-noassemble* option, because by default, the program will generate the .j files and then assemble them. With *-noassemble*, the program will not assemble the files. **To run only the syntax/semantics part of the program, comment out line 86 in *SixtyFortran.java*.**

**Object Code Generation:**

This section goes over the Dinosaur example mentioned earlier. It discusses the structure of the files generated (or at least that are supposed to be generated) by the compiler.

* *program.j*
  + This is the program that holds the main code.
* *program\_bin*
  + This folder stores all of the additional object files generated by the compiler
* *library*
  + This folder stores the precompiled code for the libraries, which includes the code for the *integer, string, bool, and real* types.
* *program­\_bin/routine.j*
  + Routines are objects, and thus each routine needs it’s own separate class. This acts as a superclass for all routine classes so that store any routine in the slot of any other routine. It also has a routine signature for all routines used in the program so that, if we’re calling a routine, we don’t need to know the routine’s class, we just need to know the method signature and we can let the JVM’s polymorphism do the rest.
    - For example, say we have *program\_bin/program$Dinosaur$printName/operator\_parenthesis()V.*

Instead of calling:

*invokevirtual* [the method above], we can just call:  
*invokevirtual Llibrary/routine/operator\_parenthesis()V* and due to polymorphism, the correct method will be run.

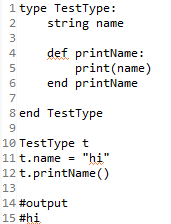
* *program\_bin/program$Dinosaur.j*
  + This represents the object file that handles the dinosaur class mentioned earlier. It has a *name* field of type *library/string,* a static field named *status* that is also a *library/string* type.
  + Since routines are objects, it also has a field for each routine and static routine, although it stores them as their superclass, *library/routine* instead.
* *program\_bin/program$Dinosaur$printName.j*
  + Routines are objects, so each routine needs its own object file. This class has a single routine called *operator\_parenthesis* that is called on the *printName* object.
    - Since this is a non-static routine, it also needs a reference to the object it is storing. Thus, the *printName* class has a field named *type* that is of type *Dinosaur*. The constructor of *printName* expects this *Dinosaur* object to be passed in as it’s one and only parameter.
    - *printName* can also access variables outside of the type. *printName* has a static field for each object that is not found locally in the routine or as a field of the type. This static field is initialized during runtime. The *print* method is an example of a nonlocal object that is statically stored by the *printName* class.
* *program\_bin/program$Dinosaur$printName.j*
  + This is similar to the *printName* class, except since it is a static routine, it doesn’t have a reference to an object of the *Dinosaur* type.

**Included Programs:**

Several programs have been included in this project that demonstrate the semantic capabilities of 64tran. Additionally, *test.f* (seen below) actually works. I have also taken your *newton.pas* program and manually converted it to 64tran. Lastly, we have included *Calculator.f,* which is a program that accepts input from the user and does some simple calculations using that input.

**What actually works:**

Not much I’m afraid. Most of the time was spent trying to get types and routines to work properly. The following program works as anticipated, and both the source and object code for the program has been included in this assignment. This program follows a similar template as the one mentioned previously, but because it doesn’t have any static fields or methods, it is actually able to run.



This demonstrates how we can define a type with a field and a routine, how we can assign the field, and how we can call a routine. It successfully prints *hi* to the console. Expressions, such as *string(1 + 2)* work as well. More features were implemented, but they didn’t work, and in fact, they ended up causing severe problems that prevented any object code from being generated at all (this was the bug we mentioned in the request for an extension). Thus, we reverted to an earlier version of the project that allows for some minor code generation. Neither *if* nor *while* work in this instance either.

The semantics pass, however, works very well. It has solid error checking, and recovery. Thus, if you want to write programs in 64tran, but not actually run them, our project allows you to do that.

**Conclusion:**

Our group was successfully able to create a good syntax and semantics pass for 64tran, and we successfully could implement a part of the compiler pass, but we simply ran out of time to implement every feature and fix every problem, especially since we all had several other major projects to work on at the same time. If we had perhaps one more week, we could get the compiler to work well on this beautifully complex language, but one more week we do not have, and thus, this is what we have to show for it.