CS3423 - Compilers 2

SPARO: Smart Pointers and ARray Operations

Project Report (Team-9)

INDEX

INTRODUCTION

SPARO is an object oriented programming language designed with inspirations from the reference model of variables in Java, smart pointers and RAII in C++, the ease for tensor and array operations in Python and the simplicity of COOL. Most object oriented features are more similar to Java and Cool(which itself is similar to Java).
The name ‘SPARO’ comes from ‘*Smart Pointers*’ and ‘*ARray Operations*’, and true to its name the SPARO language has inbuilt features which support such operations. All variables in the SPARO language are references by default and are handled using ‘*unique*’ and ‘*shared*’ pointers behind the scenes. This helps in efficient garbage collection, and is the prominent feature of the SPARO language.
As for the ‘*ARray operations*’, the language has it’s very own built in ‘*Tensor*’ class which supports convenient methods for algebraic operations. We even support special arithmetic operations on ‘*Tensor*’ objects. Few of the built in features include ‘*Tensor*’ slicing, matrix multiplication and more.
This final report describes the various features of the SPARO language and draws out its similarities and differences with other languages. This report gives a detailed description of the lexical and semantic specifications of the language. We also describe the various built in classes and methods SPARO has to offer. This report takes you through the language tutorial, the language specifications, the compiler development environment and the various phases of building the SPARO language and compiler.

LANGUAGE TUTORIAL

Now we walk you through the simple features of the SPARO language, so you can write your very own SPARO program! Sparo is an object oriented
Oriented programming language and every program consists of one or more classes. The class ‘Main’ is compulsory for each program. Within the ‘Main’ class we have our ‘main’ function where the program execution starts. With this basic background let's dive into our first SPARO program!

```
##first SPARO program!!

class Main {

     Int main(){

          “Hello world!”.put();
          “Our first SPARO program!”.put();

          return 0;
     }

}
```

And that’s it, we have written our first SPARO program! Let's quickly look at some minor details. We used the ‘put()’ function to output the string to stdout. This is a built in method supported by the ‘String’ class. At the end we make sure to return an integer value to the main function, as its return type is ‘Int’.
Now let’s go through more details of the language and see more interesting examples.

LANGUAGE REFERENCE

Main features of the SPARO language:

SPARO takes inspiration from languages such as COOL, C++ and Python, and includes concepts such as smart pointers for memory management in programs. Some built in classes are automatically available to the user, and new classes can be defined as required. However, the *Main* class must be present in every SPARO program.

Basic built in classes:

Object:

Object is the root class in the inheritance tree. It does not have any methods.

Int:

Integers are defined as non-empty strings of digits [0-9]. Negative integers have a - sign prefix.
Following are the in built methods supported by the 'Int' class:

- ***construct()*** : Constructs Int value with 0.
- ***construct(Int)*** : Constructs Int value from the argument.
- ***construct(Float)*** : Constructor initializes Int value with the Integer part of the Float's value (argument).
- ***void put()*** : Outputs the integer to stdout
- ***void get()*** : Reads the integer from stdin
- ***Bool equals(Int)*** : Checks if the **values** (NOT the pointers, unlike '==') of the Int objects being compared are equal.

Float**:**

Floating point constants consist of an optional minus(-) prefix, a string of digits [0-9], a decimal point(.) and another string of digits [0-9].
Following are the in built methods supported by the 'Float' class:

- ***construct()*** : Constructs Float value with 0.0
- ***construct(Float)*** : Constructs Float value from the argument.
- ***void put()*** : Outputs the Float value to stdout
- ***void get()*** : Reads the Float value from stdin

- ***Bool equals(Int)*** : Checks if the **values** (NOT the pointers, unlike '==') of the Float objects being compared are equal.

Bool**:**

Following are the inbuilt methods supported by the 'Bool' class:

- ***construct()*** : Constructs Boolvalue with false
- ***construct(Float)*** : Constructs Bool value from the argument.
- ***Bool equals(Int)*** : Checks if the **values** (NOT the pointers, unlike '==') of the Bool objects being compared are equal.

String:

String constants are sequences of characters enclosed in double quotes.
Following are the in built methods supported by the 'String' class:

- ***construct()*** : Constructs an empty string
- ***construct(String)*** : Constructs a string constant specified by the argument.
- ***void put()*** : Outputs the String to stdout
- ***void get()*** : Reads the String from stdin
- ***Bool equals(String)*** : Checks if the **values** (NOT the pointers, unlike '==') of the String objects being compared are equal.

Other built in classes:

Array:

Similar to C and C++, arrays in SPARO are declared by the type of elements followed by the name of the array and the dimension, which is an integer constant. Arrays can be declared as unique, shared or weak, which specifies the type of pointers that the array holds (more details can be found in the smart pointer semantics). Pointer type of the elements in the Array is always unique. The array elements, which are pointers, are initialized to a 'nullptr' by the no-argument constructor.

Tensor:

Tensors are multidimensional arrays of floating point numbers. Like arrays, the elements of a tensor are also unique pointers.All elements will be nullptr at initialization by the no-argument constructor.
Tensor class supports the following methods:

void initialize(Float): modifies the tensor object, by setting all elements to the Float constant passed as an argument.
Tensor transpose(): returns a new tensor object with dimensions 1 and 2 exchanged (i.e The rows of new object = columns of calling object, columns of new object = rows of calling object).
Note that the transpose method will throw an error if the number of dimensions of the tensor is greater than 2.
Arr<Int, num_dimensions> shape(): returns an array of Ints, whose size equals number of dimensions present in the calling tensor object.

Tensor slice([st1] : [end1], [st2] : [end2], [st3] : [end3],...., [stn] : [endn]):
returns a new tensor object such that along each dimension, elements are taken starting from 'st1' uptill (excluding) 'end1'. Here 'n' is the number of dimensions present in the calling tensor.

➔ If '*st*' is not provided, then elements starting from position '0' along that dimension are considered.
➔ If *'end'* is not provided, then all elements starting from position *'st'* till the very last element of that dimension, are considered.
➔ If *'st'* and *'end'* are both not provided, then all elements along that dimension are included in the slice.

Arithmetic, Relational and Logical Operations:

The operators used in SPARO are similar to those used in Python. The order of precedence for the operators can be found in the table below.

Precedence	**Operator**	**Description**	**Associativity**
1	f() a[] .	Function call Array/Tensor indexing Member access	Left to Right
2	- not new_unique, new_shared	Unary minus Logical not New	Right to Left
3	* @ / %	Multiplication, matrix multiplication, division, remainder	Left to Right
4	+ -	Addition and subtraction	Left to Right
5	< <= > >=	Relational/comparison operators	Left to Right
6	==	Equality comparison operator	Left to Right
7	and	Logical and	Left to Right
8	or	Logical or	Left to Right
9	=	Assignment	Right to Left
10	:	Slice range	Undefined

Smart Pointer Semantics:

Smart pointers are the characteristic feature of SPARO. The implementation of these smart pointers is similar to that in C++.

There are three kinds of pointers: *unique*, *shared*, *weak*.

If an object is pointed to by a unique pointer, only this pointer can point to it, i.e there can't be any other pointer to the same object. Whenever this pointer is copied(say a naive assignment, parameter passing etc) the pointer stops pointing to the object and instead points to *nullptr*.
When the unique pointer pointing to an object goes out of scope, the underlying object is freed.

shared pointers, unlike unique pointers, allow other shared or weak pointers to co-point to the same object. A reference counting is maintained per object, and the object is freed when the count reaches zero.

weak pointers are like shared pointers, but do not take part in reference counting.

Pointer conformance:

unique pointers conform to unique pointers and shared pointers
shared pointers conform to only shared and weak pointers.
weak pointers conform to only shared and weak pointers.

Constant semantics:

Each occurrence of 6, 8, "hello" etc are instances of a different unique pointer pointing to an object of respective type such as Int, Float etc with the desired value. Similarly intermediates on an expression are also unique pointers.

Object Orientation in SPARO

SPARO is a purely object oriented language, and supports all common object oriented concepts, such as encapsulation, inheritance and polymorphism. Specific details of OOP with SPARO can be found below.

Classes:

All code in SPARO is organized into classes. Each SPARO program consists of multiple classes that perform various functions. Class definitions have the following form:

```
class <type> [ extends <type> ] {
<feature_list>
};
```

All class names are globally visible. Class names begin with an uppercase letter. Classes may not be redefined.

Features:

The body of a class definition (*feature_list*) consists of a list of data members and member methods. Data members are variables that are a part of the objects of the current class, and member methods are functions that manipulate these variables. The names of all data members and member methods must begin with lowercase letters.

SPARO classes implement data abstraction (information hiding). Data members are accessible only within the current class and the classes that derive the current class, i.e, they have local scope. Member methods can be invoked through objects of the current class from outside the class, i.e, they have global scope.

Data members cannot be overloaded. Member methods can be overloaded as long as the argument lists are different and the return type is the same.

Data Members:

Data members are declared with an optional pointer type, followed by the data type and identifier name. The pointer type can be either *unique*, *shared* or *weak*. When the pointer type is not explicitly specified, it is declared as *unique* by default. If a data member is not explicitly initialized in the declaration statement, then it is automatically initialized to the value *nullptr*.
The syntax for the declaration of data members is:

```
[unique/shared/weak] Type identifier;
```

Member Methods:

All member methods must be defined within the class definition.
Member methods are declared with an optional pointer type, followed by the data type of the value returned by the method, the name of the method, the argument list and the method definition. Similar to the case of data members, if the pointer type is not explicitly specified, then it is declared as *unique* by default.
If a method does not return any value, then its return type must be declared as *void*.

All classes by default provide the following methods:

```
unique T copy()
shared T copy()
weak T copy()
```

where T is the class itself. The function returns the respective pointer to a newly created copy of the object.

The definitions of member methods have the following form:

```
[unique/shared/weak] <type> <id> ([<type> <id>, …])
<compound-statement>
```

EXAMPLE:

```
class Main {

    Int max;     ##'max' declared,  will be null_ptr
    shared Int findMax(Array<Int, 10> local_arr) {
        max = local_arr[0].copy();
        for(Int i = 0; i < 10; i = i+1){

            if(local_arr[i] > max){
                max = local_arr[i].copy();
            }
        }
        return max.copy();

    }
```

Constructor:

The constructor is a member method which initializes the objects of the current class. It is automatically called when an object of the current class is created. Constructor functions are declared with the keyword *construct*, followed by the argument list, the superclass constructor call and the constructor definition.
The keyword *after* is used to specify the superclass constructor to call, and the arguments to pass. When the constructor of the current class is called, these inherited constructors are called from Object all the way to the immediate super class, and then the instructions in the constructor definition are executed. Constructors can be overloaded with different signatures.
Constructor methods have the following form:

```
construct ([<type> <id>, …]) after ([<id>, …])
<compound-statement>
```

EXAMPLE:

```
class Base {
    Int x;
    ## Note that use of 'after' is necessary, as even Base
                                  class inherits from 'Object'
class.
    construct(Int a) after() {
      x = a;
    }
    void print() {
        "In the Base class!\n".put();
    }
};

class Derived extends Base {

    Int y;

    construct(Int a, Int b) after(a) {
      y = b;
    }

    void print() {
        "In the Derived class!\n".put();;
    }
};
```

Destructor:

The destructor is a member method which is used to destroy objects of the current class.
The destructor is automatically invoked when an object of the current class is freed, either when the scope of its unique pointer is lost or when there are zero reference counts by shared pointers.
The destructor method of a class does not take any arguments, and hence cannot be overloaded.
Destructor methods have the following form:

```
destruct ()
<compound-statement>
```

New objects:

New expression is used to construct a new object and is elaborated in a further section on expressions.

Inheritance:

A class can extend (i.e, inherit from) exactly one other class. If no parent class is specified, then it extends the *Object* class by default.
A class inherits all the data members and the methods from its superclass. The inheritance graph should be a tree, with the *Object* class as its root.

Member methods of a class can be overridden in a class derived from it. However, data members can not be redefined as the same or different type.

Polymorphism:

SPARO supports polymorphism two ways:

If a derived class overrides a base class method, and the method is invoked through a Base class pointer (whose value, however, is a derived class object), then the derived class method will be dispatched.
For example,

```
Base ptr = unique_new Derived();
ptr.foo();
```

will result in the execution of the “foo()” method defined in the Derived class.

Within the same class, methods can be overloaded with the same name, as long as the argument lists are different. Constructors can also be overloaded.

this:

Inside methods of class “Type”, the *this* keyword acts as if it is a data member of the type “*weak Type*” and points to the object through which the method was dispatched. *this* cannot be assigned to something else.

The *Main* class:

Every program must have a class *Main*. Furthermore, the Main class must have a method *main* that takes no formal parameters. The *main* method must be defined in class *Main* (not inherited from another class). The return type of *main* should be Int.
A SPARO program is executed by evaluating this main method, so all the instructions to be executed must be contained within this method. The return value is returned to the shell.

Imperative features

Sparo supports the following standard imperative features with similar syntax and semantics to C++.

Compound statements:

Zero or more statements inside a pair of braces.

Selection statements:

A conditional has the form:

```
if (<expression>)  <compound-statement1>
[else <compound-statement2> ]
```

The static type of the expression has to be Bool. If the expression evaluates to true, the *compound-statement1* is executed. If the expression evaluates to false, the *compound-statement2* is executed, if present.

While loop:

```
while(<expression>) <compound-statement>
```

The static type of the expression has to be Bool. The compound statement is executed repeatedly as long as the expression evaluates to true before every iteration.

For loop:

```
for([<expression1> | <declaration1>]; <expression2>;
[<expression3>] ) <compound-statement>
```

The static type of expression2 has to be Bool. The scope of declaration1 is limited to expression2, expression3 and the compound-statement.
One difference between C++ and Sparo for loop is that here expression2 is not optional.

Return statement:

```
return [<expr>];
```

Returns the expr from the function. The static type of <expr> must conform to the return type of the function. If and only if return type is void, then expr should not be there.

Break statement:

```
break;
```

Breaks out of the innermost loop. The control is resumed at the statement immediately following the innermost loop.
continue; statement is not supported.

Expressions:

Expressions in Sparo can be slice expressions, assignment expressions, logical operation expressions(or, and, not), relational operation expressions(equality, less than, less than equal, greater than, greater than equal), arithmetic expressions(addition, subtraction, multiplication, division, modular division, new expression, method/member access(through dot operator), method dispatch, an array/tensor indexing, or a primary expression.

Most of the expressions follow well known syntax/semantics similar to popular languages like C++ or Java. Some important specifics that were not previously mentioned are mentioned below:

New expressions:

```
new_unique <type>([<expr>, <expr>, …])
new_shared <type>([<expr>, <expr>, …])
```

<type> should have a constructor such that the actual arguments types conform to the respective formal arguments of the method. The value of the new expression is a unique/shared pointer respectively based on new_unique or new_shared to the <type> pointing to a newly constructed object of <type>, through the constructor.

Assignment and Equality is with pointers!

Both the assignment and equality semantics are pointers based. That is, upon assignment, the LHS pointer starts pointing to the object pointed by the pointer on the RHS. The equality compares if both the pointers point to the same object. For equality on value, the in built types support a .equals() method. Other user defined types can support similar methods.

Array/Tensor Indexing:

The array indexing is of the common syntax:
`<expr1>’[‘<expr2>’]’`
`<expr1>` should be of the Array type.
`<expr2>` should be of Int type.

Tensor indexing takes index for all the dimensions within same square brackets:
`<expr0>’[‘<expr1>, <expr2>, … <exprN>’]’`
`<expr0>` should be of Tensor type, with N dimension measures.
`<expr1>, <expr2>, …..<exprN>` should be of Int type.

The static type of the expression is the base type of the Array. For Tensor, the static type is Float.
If at runtime, the index is out of bounds, runtime error is thrown.

Integer division and modular division:

The integer division strips off the fractional part. Eg: 5 / 2 = 2, -5 / 2 = -2.
The % operator is consistent with the identity a = a/b*b + a%b.

Operand types:

Type1 <op> Type2 (if binary) <op> Type (if unary)	Result type	Types allowed
- T	T	T = Int, Float, Tensor
not Bool Bool and Bool Bool or Bool	Bool	
T + T T - T T * T T / T	T	T = Int, Float, Tensor
Int % Int	Int	
Tensor<...> @ Tensor <...>	Tensor<...>	Dimensions respecting matrix multiplication semantics
T < T T <= T T > T T >= T	Bool	T = Int, Float
T == T	Bool	T = any type

SPARO Lexical Specification

Lexical Structure

The lexical units of Sparo are constants, type identifiers, object identifiers, keywords, special symbols/operators, comments(ignored) and whitespace(ignored).

Constants

Integer constants are non-empty strings of digits 0-9. Negative integers have a - sign prefix.
‘-’?[0-9]+

Floating point constants have an optional minus(-) prefix, a string of 0-9, a decimal point(.) and a string of 0-9.
‘-’? [0-9]+ ‘.’ [0-9]+

String constants are enclosed in double quotes "...". Within a string, a sequence ‘\c’ denotes the character ‘c’, with the exception of the following:

\b	Backspace
\n	Line feed
\t	Horizontal tab
\f	Form feed

A non-escaped newline character may not appear in a string:
"This
is not OK"
"This is \
OK"

A string may not contain EOF.
A string may not contain the null.
Any other character may be included in a string.
Strings cannot cross file boundaries.
“‘‘ ('\\'~'\u0000' | ~["\n\u0000\\])* “‘‘

Bool constants are the keywords true and false.
Pointer constants is just the keyword nullptr.

Identifiers

Identifiers are strings (other than keywords) consisting of letters, digits, and the underscore character. Type identifiers begin with a capital letter; object identifiers begin with a lowercase letter.

```
OBJECTID    : [a-z][a-zA-Z0-9_]*
TYPEID      : [A-Z][a-zA-Z0-9_]*
```

Special symbols/operators:

The special syntactic symbols (e.g., parentheses, assignment operator, etc.) are:
( ) : @ ; , + - * / % < > = { } [ ] .

Comments

There are two forms of comments in SPARO. Any characters between 2 hash symbols "##" and the next newline (or EOF, if there is no next newline) are treated as comments. Comments may also be written by enclosing text in #∗ . . . ∗#.
Comments cannot be nested.
Comments cannot cross file boundaries.

Keywords

this, Int, Float, String, Bool, Tensor, void, class, extends, after, unique, shared, weak, construct, destruct, nullptr, if, else, while, for, and, or, not, true, false, new_unique, new_shared, break, return

SPARO Grammar

program:
class-definition-list(opt)

class-definition-list:
class-definition
class-definition-list class-definition

class-definition:
class-head **{** member-list(opt) **} ;**

class-head:
class type-identifier base-clause(opt)

base-clause:
extends class-name

member-list:
member
member-list member

member:
member-declaration **;**
constructor
destructor
method-definition

destructor:
destruct ( ) compound-statement

constructor:
construct (parameter-declaration-list(opt) **) after (** expression-list(opt) **)**
compound-statement

method-definition:
declaration-specifier object-identifier **(** parameter-declaration-list(opt) **)**
compound-statement

parameter-declaration-list:
parameter-declaration
parameter-declaration-list **,** parameter-declaration

parameter-declaration:
declaration-specifier object-identifier

statement :
declaration **;**
expression-statement **;**
compound-statement
selection-statement
iteration-statement
jump-statement **;**

member-declaration:
declaration-specifier object-identifier **;**

declaration :
declaration-specifier init-declarator

declaration-specifier:
pointer-type(opt) type-specifier

type-specifier:
Int
Float
Bool
String
void
array-specifier
tensor-specifier
type-identifier

array-specifier:
Array < type-specifier **,** integer-constant **>**

tensor-specifier:
Tensor < integer-constant-list **>**

integer-constant-list:
integer-constant
integer-constant-list **,** integer-constant

pointer-type:
unique
shared
weak

init-declarator:
object-identifier
object-identifier **=** expression

expression-statement:
expression(opt)

expression-list:
expression
expression-list **,** expression

expression:
slice-expression

slice-expression:
assignment-expression(opt) **:** assignment-expression(opt)
assignment-expression

assignment-expression:
logical-OR-expression
unary-expression **=** assignment-expression

logical-OR-expression:
logical-AND-expression
logical-OR-expression **or** logical-AND-expression

logical-AND-expression:
equality-expression
logical-AND-expression **and** equality-expression

equality-expression:
relational-expression
equality-expression **==** relational-expression

relational-expression:
additive-expression
relational-expression **<** additive-expression
relational-expression **>** additive-expression
relational-expression **<=** additive-expression
relational-expression **>=** additive-expression

additive-expression:
multiplicative-expression
additive-expression **+** multiplicative-expression
additive-expression **-** multiplicative-expression

multiplicative-expression:
unary-expression
multiplicative-expression ***** unary-expression
multiplicative-expression **@** unary-expression
multiplicative-expression **/** unary-expression
multiplicative-expression **%** unary-expression

unary-expression:
secondary-expression
new-expression
unary-operator unary-expression

unary-operator:
-
not

new-expression:
new-keyword type-specifier **(** expression-list(opt) **)**

new-keyword:
new_unique
new_shared

secondary-expression:
primary-expression
secondary-expression **[** expression-list **]**
secondary-expression **(** expression-list(opt) **)**
secondary-expression **.** object-identifier

primary-expression:
this
object-identifier
constant
(expression **)**

constant:
integer-constant
string-constant
floating-constant
boolean-constant
nullptr

compound-statement:
{ statement-list(opt) **}**

statement-list:
statement
statement-list statement

selection-statement:
if (expression **)** compound-statement
if (expression **)** compound-statement **else** compound-statement

iteration-statement:
while (expression **)** compound-statement
for (for-initializer(opt) **;** expression **;** expression(opt) **)** compound-statement

for-initializer:
declaration
expression

jump-statement:
break
return expression(opt)

PROJECT PLAN

This section describes our project timeline. The project managers of our team helped schedule meetings to plan and complete different phases of the project.

Phase 1 : Language Design	Basic language design of SPARO. Deciding on what features to incorporate and what to ignore. Studied various other languages, like Java, C++, Python to finalize our language features. Decided that smart pointers and tensor operations would be the prominent feature of our language
Phase 2 : ANTLR Tool	Next we familiarized ourselves with the working of the ANTLR tool. We went through the Github documentation of ANTLR, and learnt how to use the tool to start building the compiler for SPARO
Phase 3: Lexical analysis	Here we started writing the first phase of our compiler, the lexical analyser. We used the support of regular expressions in ANTLR to form tokens of our SPARO source code. Basic error handling features were also added.
Phase 4: Parsing	As part of parsing our main task was to formulate and write the grammar for our language. We referred to the c++ reference manual and kernighan ritchie for writing the grammar. Once we completed writing our formal grammar we wrote the same as an ANTLR 'g4' file. By the end of this phase we were able to generate a valid Parse tree for our language.

Phase 5: Semantic Analysis	This phase of building the compiler involved writing a lot of code to ‘walk’ the parse tree and perform various semantic checks. By the end of this phase we were able to generate helpful error messages for incorrect programs.
Phase 6 : Final phase, Documentation	As the final phase of the project, we spent time commenting and documenting our code. We reported all of our language specifications and implementation details.

LANGUAGE EVOLUTION

The language SPARO is not in the same form as it started. Since its inception, a number of features have been added/removed considering various factors like ease of parsing, ease of code generation, available time etc.

The initial idea was to support a language as general purpose as possible. For simplicity, it was assumed that being similar to COOL will help. As a characteristic feature, we decided that while we follow the reference model of variables similar to COOL/Java etc, we need to get done away with their garbage collection and enforce smart pointers instead. This embrace of smart pointers and RAII led us to fall closer to C++ model of OOP with more features such as overloaded constructors and destructors. We also decided to have a strict and static type system. In order to make our language convenient and as an attraction feature, we decided to support special syntaxes for array and tensor operations as supported in Numpy arrays etc.

Features such as list comprehensions, compile-time type inference, wider support for complex data types as part of the standard libraries, were initially fancied. However they were let go due to the growing complexity of the language.

Some other features underwent modifications for the sake of implementation-friendliness. For example, initially we decided that braces are optional for single-statement if blocks, loop blocks etc. However to prevent the dangling else ambiguity, the language enforced that for if-else blocks, braces are mandatory. Comments are not allowed to be nested in order to support the greedy lexers/parsers.

It was thought to support multiple inheritance, or support interfaces similar to Java. However, that would require things like vtable in C++ to support polymorphism. Hence it was decided to make every class to inherit from exactly one parent class.

Another feature that dragged a good amount of discussion was to differentiate between Arrays and Tensors. Initially it started out as them both being the same, except for Tensors supporting additional fancy syntax like numpy. Later however we decided that while multidimensional arrays will be seen as array of pointers to arrays, a multidimensional tensor would occupy a contiguous space in the memory. This is also reflected in the syntax where a 5x5 Tensor would be declared as `Tensor<5, 5>` (the base type is `Float` only for tensors), but the same matrix if as an array would be declared as `Array<Array<Float, 5>, 5>`.

COMPILER ARCHITECTURE

Most of the SPARO compiler features, such as Lexical Analysis, Syntax Analysis and Semantic Analysis have been developed using the ANTLR parser generator. The specifics of each stage of compilation are explained below.

Lexical Analysis

Lexical Analysis has been done using ANTLR to precisely convert SPARO programs into their corresponding tokenized forms. The SPARO lexer contains complex rules for analyzing different kinds of tokens and identifying and reporting any lexical errors in the programs.

Syntax Analysis

Syntax Analysis in the SPARO compiler has been implemented using a left recursive grammar defined using ANTLR. A set of concise rules have been defined to elegantly capture intricate details of the syntactic structure of SPARO programs. Syntax errors are accurately identified, and error recovery is performed as supported by ANTLR. The interface with the lexical analyzer is also antlr generated, where the tokens stream output by the lexer is passed on as input to the parser. Similarly the rule context generated for the root rule(program) is then passed as a ProgramContext object to the semantic analyzer.

Semantic Analysis

Semantic Analysis has been done using the parse tree generated by ANTLR after syntax analysis. For the sake of clarity and modularity, the grammar itself is not annotated; instead, the parse tree generated is traversed separately. The semantic analyzer supports various features, such as the detection of cycles in the inheritance graph and detection of redeclarations of classes, methods and members. Error messages are shown along with line numbers.

DEVELOPMENT ENVIRONMENT

Various tools and practices were adopted to ensure smooth development and integration throughout the project. Some of them are reported below:

For version control, a private Github repository was maintained for the codes. Prior to that, during the language design ideation phase, Google Docs were maintained to reliably save our ideas and to easily collaborate.

For convenience, VS Code was used as the preferred text-editor. Coupled with Java extension and adding ANTLR jar file as an external library, the Intellisense autocomplete was very helpful to make full use of the ANTLR generated parse trees. This was especially helpful during semantic analysis.

The build process involves two stages:

1. Generation of java source files from .g4 files of lexer and parser. This is done using ANTLR4. A convenient Makefile is made in the `src/grammar` directory. The output java files are sent to the `/src/java` directory.

2. Compilation of ANTLR generated java files of SparoLexer, SparoParser and R RuleContext classes, and our own files such as Semantic.java for semantic analysis. These have to be compiled together with ANTLR4 library. A different Makefile is used for the same.

To put both the above processes together, a Makefile is present in the `/src` directory itself, which would sequentially call the Makefiles for step 1 and step 2. Thus, all it takes to build at the end is just executing `$ make` from `/src` directory.

TEST PLAN AND TEST SUITES

All testing related scripts are present in the `/src/test` directory. Before testing, we need to automate the generation of meaningful outputs from the command line with minimal clutter. Following the ideas of the testing suite present in the COOL project templates and our Compiler - 1 assignments, we created the boilerplate classes LexerTest, and SparoSemanticTest respectively. For parser, we make use of ANTLR4's TestRig tool. All sample programs/test programs are in the `/Sample_programs` directory. We elaborate on test suite for each phase of the compiler below:

The class LexerTest is written similar to the class given as part of Compiler-1 assignment. This runs the ANTLR generated SparoLexer on the given input program. It prints the tokens and errors that were lexed along with the corresponding text and line numbers. The ANTLR4's TestRig has two modes, a GUI mode that prints a neat parse tree to inspect manually, and a text output of the parse tree generated. The SparoSemanticTest prints the errors in the stdout and other logs on what it analyzed in semanticlogs.txt file.

The bash scripts `/test/lexer.sh`, `/test/parser.sh`, `/test/gui-parser.sh`, and `/test/semantic.sh` are convenient scripts to fire up LexerTest, ANTLR4 TestRig for parse trees in text mode, in GUI mode and SparoSemanticTest respectively.

Whenever a new sample program is added, we run the `/test/lexer.sh` and `/test/gui-parser.sh` script files and test the output manually. Further on, this can be automated with the `/test/add-test.sh`. This script runs the above tools, and adds the output of lexer.sh and parser.sh into the `/test/lexer-ans/` and `/test/parser-ans/` directories respectively. These serve as the expected outputs.

Then whenever any changes to the lexer is made, running the test scripts `/test/auto-lexer.sh` runs `/test/lexer.sh` on every file in `/Sample_program/` directory and compares the output with expected output present in `/test/lexer-ans/` directory and reports any discrepancies. An analogous mechanism for the parser is supported through the `/test/auto-parser.sh` script, which looks for expected outputs in `/test/parser-ans/` directory.

For semantic analysis, as of now only manual inspection after running `/test/semantic.sh` is supported.

CONCLUSIONS

Our vision was to make an object oriented language which implemented advanced concepts like smart pointers and tensor operations. We stressed on the importance of dynamic memory management without the problem of slow garbage collection.

We aimed to build a compiler that could efficiently convert SPARO programs into optimized machine language, and we attempted to do so using ANTLR's powerful parser generating capability integrated with LLVM's code conversion.

However, we were not able to complete the project as we promised in the beginning. Some of the things that we were unable to present was Intermediate code generation and optimization.

Lessons Learned

- We should not be too ambitious while designing the language.
- It is important to plan goals and deadlines in advance.
- Many language features exist just to make implementation easy.
- Underestimated time taken for implementation, should include buffer time in our schedule to deal with unexpected errors.
- Although we struggled from time to time, designing the compiler was enjoyable and insightful.

We learned a lot throughout the course and we will definitely try to apply the knowledge we gained to other applications that we design in the future.

APPENDIX

The github link for the code repository:

https://github.com/IITH-COMPILERS2/compilers2-project-team-9

Sources used to write the documentation:

- The Java Whitepaper - https://www.stroustrup.com/1995_Java_whitepaper.pdf
- The C Programming Language Book by Kernighan and Ritchie

Sources used to implement the various parts of the compiler:

- Official ANTLR documentation - https://github.com/antlr/antlr4/blob/master/doc/index.md
- The Definitive ANTLR 4 Reference, 2nd Edition
- C++ Reference Manual - https://www.stroustrup.com/C++.html
- The C Programming Language Book by Kernighan and Ritchie
- Principles of Compiler Design course on NPTEL - https://nptel.ac.in/courses/106/108/106108113/
- Mini Java Compiler by csaroff - https://github.com/csaroff/MiniJava-Compiler
- COOL Compiler Assignments Templates
- The xtensor library - https://xtensor.readthedocs.io/en/latest/
- Smart Pointers in C++ - https://docs.microsoft.com/en-us/cpp/cpp/smart-pointers-modern-cpp?view=msvc-160