Department of Informatics, University of Leicester

CO3015 Computer Science Project

Dissertation

Compiler Constructions for an Object Oriented Programming Language

Michael Lynch
May 2017

DECLARATION

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or submitted at any stages, which are taken from other people's work have been specifically acknowledged by clear citation of the source, specifying author, work, date and page(s).
Any part of my own written work, or software coding, which is substantially based upon other people's work, is duly accompanied by clear citation of the source, specifying author, work, date and page(s).
I understand that failure to do this amounts to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

Name: Michael Lynch

Signed:

Date: 24th May 2016

The purpose of this project is to produce an object oriented programming language that can be used by a programmer for the purposes of producing a branching narrative. The language is capable of mitigating the circumstance that often arise in video games that feature such a narrative, when a story character goes missing in a time when they are required for a story event. Where a traditional video game narrative system may simply break and not allow the player to continue on this narrative path, this language can allow a programmer to replace missing characters intelligently to ensure that the player doesn't lose the opportunity to see what happens next.

Contents

1 Introduction

The aim of this project was to create a compiler for a special purpose object oriented programming language.
This language can be used to generate a data structure of a complex branching story that even with dramatic changes in the events and characters does not become broken and disjointed.
Many story/quest systems found in video games of today directly link the characters to the events that take place in the story. This means that should this given character die before the event takes place, an entire section of the story maybe cut short.
In order to counter this problem these characters are given so called "plot armour", effectively making them immortal only
for the purposes of the story and therefore breaking immersion for the player.
With this project I am providing an alternative method to solving this very common problem. Rather than linking the characters directly
to the events in the story this programming language allows the user to better describe why this character takes part in an event
and therefore a suitable replacement character can take over should the original character not make it.
The compiler produces as output an XML file holding information on how the events of the story unfold that can be repurposed by other programs as they see fit.
It contains features that are common to many other compilers including pre process statements allowing for multiple files to be included
into the source code as well as error messages with line numbers provided to correct the users mistakes.

2 Requirements

2.1 Output Requirements

~~O1~~ The program must be able to directly print it's output into the command line
This requirement was not followed. Given the high levels of data that appears in the output of the compiler, I decided that printing out to the command line was simply not a viable option as as it appeared way too clutered in the terminal screen to ever be useful in practice. This was not a highly important requirement as the main use of this program is for the output to be repurposed by other programs to use and as such the XML output would be much more suitable.

O2 The program must be able to print it's output into a text file that is specified by the user

O3 The output will contain the final state of the variables in the object as well as one extra output which, taking account of the objects dependencies decides whether this object existed in the story or not (see section 4.2)

2.2 Language Requirements

L1 The program will have an in-built class called Entity that is used to represent objects or subjects in a story

L2 The program will have an in-built class called Event that is used to represent significant points in the story

L3 The user will be able to create classes. These classes are used to represent the Entities and Events of the story and therefore are always derived from the 'Event' or 'Entity' class or their children

L4 The user can include other source files into the current file to get access to the classes and objects that have been created there

L5 When extending a class, users should be able to override methods that already exist

L6 Classes will have a special constructor method that will run upon instantiation of an object of this type

L7 The programming language will have a built-in function called 'println' which will print a line to the console during the compilation of the program

L8 The programming language will have a built-in function called 'GetEventContainingDataValue' which will point to an Event object that satisfies a certain condition (see section 4.1)

L9 The Event class will have an abstract special function called 'onEvent' which is called when the Event is triggered in the story

L10 The Event class will have an integer variable called 'eventYear'. This is the date in which this event takes place and affects when the 'onEvent' function is triggered

L11 The Entity class will have two integer variables called 'startYear' and 'endYear'. This variables correspond to the dates that an Entity starts existing in the story and stops existing (It's birth and death date)

L12 The programming language must support data types of Object, Integer, Float, String, Array and String
Although all of these data types do exist in the program in some manner, Arrays are only used for holding the dependencies of user created Objects. I decided early on that this feature was not going to be added as it would have required a significant amount of time dedicated to it's implementation despite not being particularly crucial in acheiving the main aims of this project.

L13 When instantiating objects the user must give the objects that it is dependent on for it to exist/take place (see section 4.2)

L14 The semicolon will be used to indicate the end of a program line

2.3 Parser Requirements

P1 Although There is a line ending character (';'), the parser will take account of real line endings when outputting to the user which line a syntax error occured on

P2 If there are syntax errors in the user's source code then the parser must be able to give the line number of where the earliest syntax error occured. The parser will then not attempt to continue compiling (stopping it from finding subsequent errors).

P3 Before parsing starts, the program will check to see whether the user has included any other source files into this source files. If they have then the compiler will load these files.

P4 Loaded source files will have their 'include' statements removed and then their source code appended to each other. The parser will then attempt to parse the code as a single document.

P5 When appending documents their must be a seperation string so the parser knows when one file has finished being parsed and the next file is starting. This is used so that in case the parser finds a syntax error it will be able to provide the line number and the correct file for where the error took place

2.4 Technical Requirements

T1 The program will assess object dependencies in the order in which the Events and Entities take place (Object dependencies will be assessed at the time of with the Event takes place given in the 'eventDate' variable and the 'entityStartDate' for Entities)

3 Timeline Programming Language

3.1 Backus Naur Form

Back Naur Form uses format from the Gold parser [3]

```
1  Sets
2  {Ident header} = [a..z, A..Z]
3  {Ident tail} = {Ident header} | {Digits} | [ ]
4  {String chars} = {All printable characters} - ["]
5  {Digits} = [0..9]
6
7  Terminals
8  Identifier = {Ident header}{Ident tail}*
9  StringLiteral = '"' {String chars}* '"'
10 Integer = {Digits}*
11 Float = {Digits}* | {Digits}{Digits}* '.' {Digits}{Digits}*
12 OperatorSym = [*/-+]
13 CommentStartSym = '<'
14 CommentEndSym = '>'
15
16 NonTerminals
17 <program> ::= <code block> |
18   <code block> <program> |
19   <class definition> |
20   <class definition> <program>
21 <code block> ::= <line> |
```

```
22   <line> <code block>
23 <line> ::= <statement> ';'
24 <statement> ::= <variable definition> |
25   <variable initiation> |
26   <function call>
27 <variable initiation> ::= identifier identifier
28 <variable definition> ::= <variable initiation> '=' <expression> |
29   identifier '=' <expression> |
30   <sub variable identifier> '=' <expression>
31 <expression> ::= <subexpression> |
32   <object definition> |
33   <equation> |
34   <array immediate definition>
35 <sub expression> ::= Identifier |
36   Integer |
37   Float |
38   StringLiteral |
39   <Sub variable identifier> |
40   <Function call>
41 <equation> ::=  <sub expression> Operator <sub expression> |
42   <sub expression> Operator <equation>
43 <object definition> ::=
44   'new' <function call> <array immediate definition>
45 <function call> ::= identifier '(' <parameter setting statement> ')' |
46   identifier '(' ')'
47 <parameter setting statement> ::= <last parameter setting> |
48   <not last parameter setting> <parameter setting statement>
49 <last parameter setting> ::= <expression>
50 <not last parameter setting> ::= <expression> ','
51 <sub variable identifier> ::= <possible object> '.' <possible object> |
52   <possible object> '.' <sub variable identifier>
53 <possible object> ::= identifier |
54   <function call>
55 <array immediate definition> ::= '[' <parameter setting statement> ']'
56 <class definition> ::=
57   'class' identifier 'extends' identifier '{' <class code block> '}' |
58   'class' identifier 'extends' identifier '{' '}'
59 <class code block> ::= <code block> |
60   <code block> <class code block> |
61   <method definition> |
62   <method definition> <class code block> |
63   <construction definition> |
64   <construction definition> <class code block>
65 <method definition> ::=
66   identifier identifier '(' <parameter definition statement> ')'
67     '{' <code block> '}' |
68   identifier identifier '(' <parameter definition statement> ')' '{' '}'
69 <parameter definition statement> ::= 'void' |
```

```
70    <parameter definition statement filled>
71  <parameter definition statement filled> ::=
72    <last parameter definition> |
73    <not last parameter definition>
74      <parameter definition statement filled>
75  <last parameter definition> ::= <variable initiation>
76  <not last parameter definition> ::= <variable initation> ','
77  <construction definition> ::=
78    '_construct' '(' <parameter definition statement> ')'
79      '{' <code block> '}' |
80    '_construct' '(' <parameter definition statement> ')' '{' '}'
```

3.2 Introduction

- The Timeline Programming Language is a tool for generating the outcome of a branching story given the data given by the user.
- The user can represent the story using Entities (used for people or objects who are in the story for a set period of time) and Events (take place once changing certain aspects of other objects in some way when it happens)
- The users input is compiled into the resulting story in an XML for further use

3.3 Example TPL Document

TPL source code

```
class Person extends Entity {

  _construct(int birthYear, int deathYear, string nameP) {
    name = nameP;
    startYear = birthYear;
    endYear = deathYear;
  }
}

Person zac = new Person(1950, 2030,"Zac")[];
Person michael = new Person(1996, 2070, "Michael")[zac];
```

Resulting XML output

```
<?xml version="1.0" encoding="UTF-8"?>
<timeline>
  <events>
  </events>
  <entities>
    <object type="Person" happened="true" id="a3df60">
      <dependencies>
      </dependencies>
      <variables>
        <var type="string" name="name">Zac</var>
        <var type="int" name="startYear">1950</var>
        <var type="int" name="endYear">2030</var>
      </variables>
    </object>
    <object type="Person" happened="true" id="a3eb90">
      <dependencies>
        <object-pointer id="a3df60"/>
      </dependencies>
```

```
19        <variables>
20          <var type="string" name="name">Michael</var>
21          <var type="int" name="startYear">1996</var>
22          <var type="int" name="endYear">2070</var>
23        </variables>
24      </object>
25    </entities>
26  </timeline>
```

Interpreting the Output This relatively simple program produces an XML document that holds key information about what happened during it's execution.

- All objects that the user has defined in the source code appears in the output file along with the state of their variables.
- The program also gives information on whether an Entity or Event exists in the story that has been generated (the 'happened' attribute on each object).
- Each object has been assigned an id in which it is referred to by other objects (object pointer using id 'a3df60' on line 15 of the XML file to refer to "Zac" in the dependencies for "Michael").
- All output files split up objects into either being Entities or Events according to the class of the object that was provided.
  In this case, Person was extended from the Entity class (see line 1 of the TPL source code) and for that reason the Person objects appear in the Entities list in the XML output (line 5-25).

3.4 Understanding Events and Entities

All objects in this language represent some sort of meaning in the story being generated and as such fit into two basic categories, Events and Entities.

As such, all classes that are created by the user must derive from the Event or Entity base classes, or child classes of that type.

An Event represents a moment in the story where some sort of action takes place. This may for example be a character in the story travelling from one location to another or someone being elected mayor of a town.

An Entity represents something tangible that exists in the world. This may be a building or an animal. Entities exist in the world for a certain period of time and then disappear just like objects in the real world do.

Event

The Event class has two variables,
The string variable 'name',
And the integer variable 'eventYear'.
'eventYear' holds the year in which this Event takes place. This effects the order in which it is processed by the timeline.

The Event class also has a special purpose method. The 'onEvent' method. This method runs while the timeline is being simulated and is allows the Event to manipulate other objects if the user chooses to override the method.

Entity

The Entity class has three variables,
The string variable 'name',
And the integer variables 'startYear' and 'endYear'.
'startYear' holds the year in which the Entity first appears in the world. For a building it would represent when it is built.
'endYear' holds the year in which the Entity stopped existing in the world. This would be a building's demolition.

3.5 Object Dependency and the Timeline

This system is designed for generating the outcome of a branching narrative and as such a timeline of the order of events needs to be simulated in order to understand how the story unfolds.
As with all stories. The existence of some events or entities depend on the existence of other events and entities. As an example, your parents cannot get mad at you for not cleaning up after a party, if you didn't have the party. This makes the Event of your parents getting mad depend on the Event of you throwing a party. The party may in turn depend the 'life

of the party', Zac, turning up.

Objects can be made dependent on each other during their instantiation by adding the object variable to array following the constructor call (This assumes the existence of two classes derived from Event, 'Party' and 'Anger'):

```
1         Person zac = new Person(1990, 2060, "Zac")[];
2         Party party = new Party(2010,"Great New Year's Party")[zac];
3         Anger parentAnger = new Anger(2011,"Parent's get angry")[party];
4
```

Each object in the timeline is processed, in order, to check that it is possible for it to take place.

In this example code the compiler first tests the 'zac' object in the year 1990.

'zac' does not depend on any other object therefore 'zac' does exist between the years 1990 and 2060.

'party' depends on the existence of 'zac', in 2010 'zac' does exist therefore the party takes place in the year 2010.

Should 'party' have taken place after 2060 it would have meant that it would not have taken place as 'zac' would not have been able to make it to the party.

Finally the compiler checks if 'parentAnger'. 'parentAnger' depends on the 'party' that did happen in the previous year, this therefore means that 'parentAnger' also takes place.

Should 'parentAnger' been set to happen in the year 2009 it would have meant that it could not take as the 'party' that it depends on has not happened yet.

3.6 The GetEventContainingDataValue Function

This in-built function is used to find an Event that satisfies a certain criteria. This criteria is that a chosen variable of the Event holds a certain data value which is specified by the user. The purpose of this function is to give the user greater control over why a certain event might take place.

The function has the following syntax:

```
1         function GetEventContainingDataValue(
2                   string ClassName,
3                   string ClassVariable,
4                   DataValue DataValueVariableMustHold)
5
```

ClassName is the name of the class that the Event type should be.
ClassVariable is the variable in that class that needs to compared against.
DataValueVariableMustHold is the value that the variable needs to hold in order to consider an Event correct for this function.

This function is key in making the story resilient from otherwise story breaking character deaths or unforseen plot points.

Building a Resilient Story

The following example illustrates the possible problems that this function aims to fix in a narrative: There are two classes to be used in the following examples. The first is the Person class that was defined in a previous example code in this section. The next is the 'MurderEvent' class:

```
1         class MurderEvent extends Event {
2           Person murderer;
3           Person victim;
4
5           _construct(Person murdererP,
6                      Person victimP,
7                      int eventYearP,
8                      string nameP) {
9             murderer = murdererP;
10            victim = victimP;
11            eventYear = eventYearP;
12            name = nameP;
13          }
14
15          empty onEvent(void) {
16            victim.endYear = eventYear;
17          }
18        }
19
```

This class effectively ends the life of the victim when the event takes place as it sets the victim's endYear to be the same as the MurderEvent's eventYear variable.

In the following program the user is attempting to convey that the character clare murders ellie in revenge for killing becca.

```
1         Person jenny = new Person(1900, 1950, "Jenny")[];
2         Person ellie = new Person(1890, 1970, "Ellie")[];
3         Person clare = new Person(1920, 2000, "Clare")[];
4         Person becca = new Person(1880, 1970, "Becca")[];
5
6         MurderEvent beccaMurder = new MurderEvent(ellie, becca,
7                                     1940, "Becca's murder by ellie")
```

```
                                        [ellie, becca];

        MurderEvent revengeMurder = new MurderEvent(clare, ellie,
                                          1945, "The revenge")
                                          [clare, ellie];

```

This story will run in a satisfactory manner. With a slight change to the events taking place however, the motivations of the characters becomes confusing.

```

        MurderEvent beccaMurder = new MurderEvent(jenny, becca,
                                        1925, "Becca's murder by jenny")
                                        [jenny, becca];

        MurderEvent beccaMurder2 = new MurderEvent(ellie, becca,
                                        1940, "Becca's murder by ellie")
                                        [ellie, becca];

        MurderEvent revengeMurder = new MurderEvent(clare, ellie,
                                          1945, "The revenge")
                                          [clare, ellie];

```

In this altered scenario clare still kills ellie out of revenge, however ellie never got a chance to kill becca as jenny got their first. This would generate a narrative mistake.

Now instead consider the following way of writing this story using the GetEventContainingDataValue function:

```

        MurderEvent beccaMurder = new MurderEvent(jenny, becca,
                                        1925, "Becca's murder by jenny")
                                        [jenny, becca];

        MurderEvent beccaMurder2 = new MurderEvent(ellie, becca,
                                        1940, "Becca's murder by ellie")
                                        [ellie, becca];

        MurderEvent originalMurder =
                        GetEventContainingDataValue(
                                "MurderEvent",
                                "victim",
                                becca);

        MurderEvent revengeMurder = new MurderEvent(clare,
                                          originalMurder.murderer,
                                          1945,
                                          "The revenge")
```

```
20                                                  [clare,
21                                                    originalMurder.murderer];
22
```

This new way of writing the story means that 'revengeMurder' is now not linked to specific possible killers of becca and instead directly links to the murderer of becca as was the original intention for the story.

3.7 Keywords, data types, and other in-built functions

Keywords

- <comment>: Triangle brackets are used to make comments in the code
- new: Used to instantiate new objects.

```
1       Bar foo = new Bar(1900, "FooBar")[];
```

- void: Used for writing methods that do not have any parameters

```
1       string noParam(void) {
2         return("This has no parameters");
3       }
```

- empty: For writing methods that do not return a value

```
1       empty nothing(void) {
2         <the method does nothing>
3       }
```

- class: For defining a new class
- extends: To inform what this new class is based on

```
1       class Foo extends Bar {
2         <Foo now has all the methods and variables that Bar has>
3       }
```

Data Types

- string: Used to represent an array of characters

- int: Represents a 32 bit integer number
- float: Represents a 32 bit floating point number

In-Built Functions

- println(string): Used to print out the contents of the parameter to the console followed by a newline during compilation of the program. This is mainly used for the purposes of debugging as the printout does not appear in the output file.
- print(string): Used to print out the contents of the parameter to the console with no newline added during compilation of the program. This is mainly used for the purposes of debugging as the printout does not appear in the output file.
- return(any): Used to return data from a method that requires data output.
- super(...): Used to call the constructor of the parent class while in current class' constructor

3.8 Error Reference

1. Constructor Overloading. This error will appear if a defined class has more than one constructor defined.
2. Incorrect Parameter Quantity. This error appears when a function or method that is called in the code has the incorrect number of parameters.
3. Type Mismatch. In cases where the types of a variable and a value that it is attempted to be defined as are not the same, this is the error that will occur
4. Operator Mismatch. This occurs when an equation holds values that cannot handle particular operators being given by the user.

Usually this is caused by using an operator other than the plus sign during string concatenation.

5. Illegal Equation Type. When writing an equation, if a type is used that is not valid for that particular equation, this is the error that will appear.

6. Division by 0. When an equation contains a division by 0 this error will occur to remind the user that this is impossible.

7. Operator Order Warning. Due to this compiler not using the BIDMAS rules of operator order, this warning serves as a reminder to the user to check that they have ordered the equation correctly. This is not an error like the others listed.

8. Non Existent Method/Function. If a function or method is called that does not exist This will be the error code given.

9. No Constructor Given. If the user attempts to call the super constructor for a class that doesn't have a constructor, this error code will be given.

10. Non Existent Class. This error occurs when the user attempts to reference a class that has not been written.

11. Non Existent Variable. This error occurs when the user attempts to reference a variable that does not exist.

12. Cannot Return. This error is thrown if the user writes the return function into a method that does not return a value.

13. Abstract Class Instantiation. This error will appear when the user attempts to instantiate an instance of the Event of Entity class. This is an illegal action under this language.

14. Local Object Instantiation. This error occurs when the user attempts to instantiate an object inside another class. All objects in this language must be written outside classes in the main program

15. No Sub Variables. Where the user has attempted to access sub variables of a variable or function that doesn't have any, this is the error that occurs.

16. Function Redefinition. This is an error that appears when the user attempts to rewrite a function during program execution.

17. Dynamic Value in Program Execution. If a dynamic value (A value generated by the GetEventContainingDataValue function) is attempted to be accessed before the timeline is simulated, this error will occur.

18. Variable Already Defined. This occurs when the user attempts to re-initiate a variable.

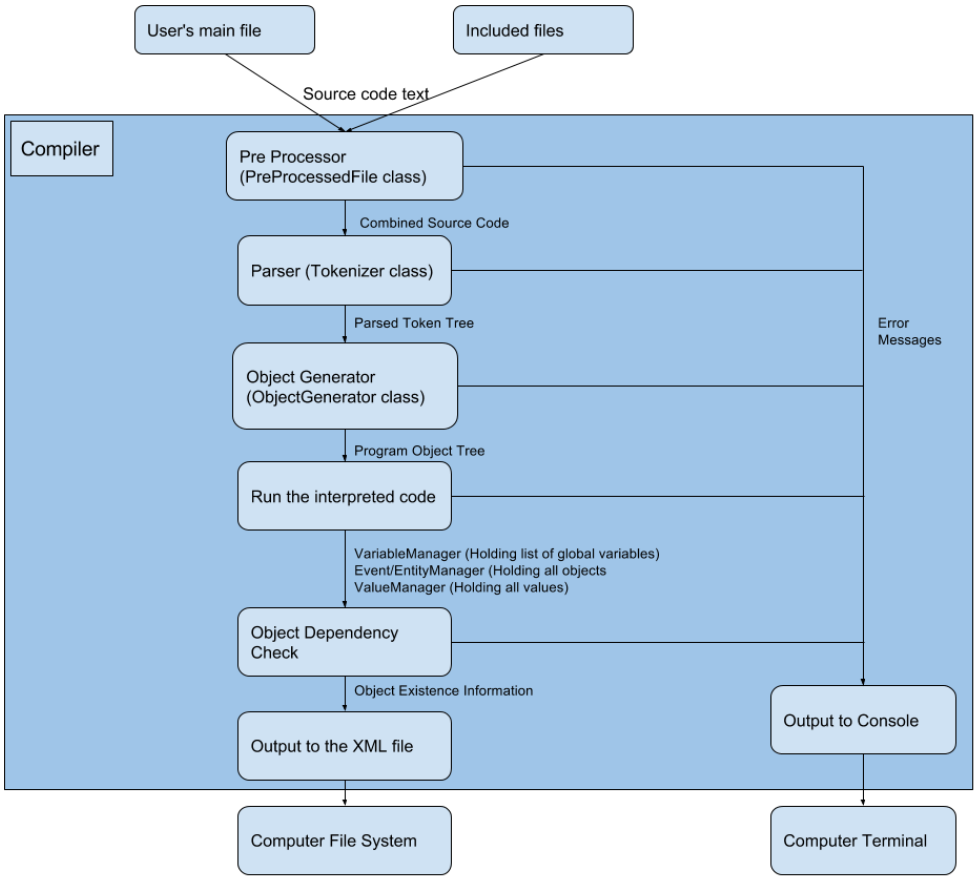
19. Parser Failure. This is caused by a syntax error in the code.

20. Pre Process Failure. This is caused by a problem with the syntax or the chosen files of an include statement.

21. No Event Available. If an attempt to access the contents of a dynamic value reveals that there are no events that fit the given requirements given in the GetEventContainingDataValue function then this error will occur. Although considered an error, sometimes code written like this might be necessary and as such the compiler always continues when finding this error.

4 Software Architecture

This program uses a pipline archtecture feeding the user's original code through several filters to create useful output at the end of the execution.



4.1 Pre Processor Component

The pre processor takes include statements from the users main source file and includes the source files that have been referenced. All of the text in these files are added to the main source code string along with markers in an array to show where one file ends and another starts. These markers are used by the program's error system in order to give the user information on which line their mistake was made on.

4.2 Parser Component

The parser uses the Recursive Depth Parser[2] algorithm to convert the user's code into a tree of tokens representing the structure of the program.

4.3 Object Generator Component

The Object Generator takes the tree of tokens constructed by the parser and converts them into a tree of objects that have a specific format. Where a Token object just held information on what type it was, what rule it was using and an array of subtokens, the objects generated here are specific to each token (there is a seperate Class for every type of token (Program, Line etc..).
This allows for more specific functionality. For example, the CodeBlock and VariableInitiation have a method called runCode(). In the CodeBlock class this method will execute all the lines held inside the CodeBlock. In VariableInitiation however, the method will add a variable to the VariableManager.

4.4 Interpreted Code Run Component

This is where the user's code is executed. This will go through the code line by line and then carrying out the required operation.
This will fill up the VariableManager, ValueManager and Event/Entity Managers with variables data and objects respectively to be used in the next phase of the system (Object Dependency Checking).

4.5 Object Dependency Checking Component

Once the user's code has been executed, The VariableManager, ValueManager and Object Managers will be filled with all the information required in order to determine what should take place in the story's timeline. This component will output whether Entities or Events that the user has defined actually exist in the story.
During this process the onEvent code of Events that take place in the timeline are run influencing what happens to Entities and Events that run later on in the story.
Once this component has finished, all the information required for outputting to the used has been generated. Information contained in the two object managers (EventManager and EntityManager) is now outputted into an XML file and the program finishes.

5 Algorithms, Data Stuctures and Software Implementation

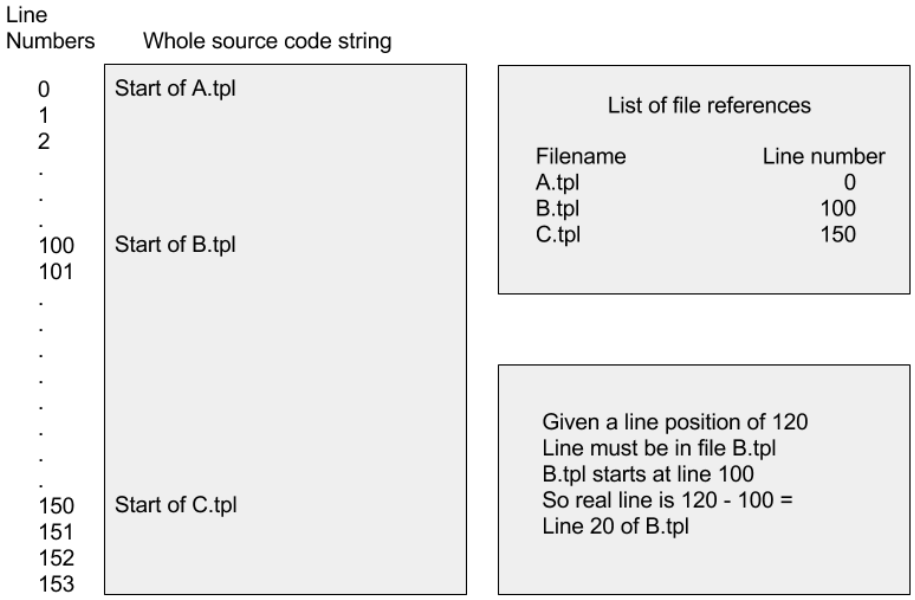
5.1 Pre Processing and Finding The Code That Produces an Error

The pre processor is what allows the user to import multiple files into their work making it easier to structure their code.
This is done by using the pre processor statement:

```
1       #include "filename"
2
```

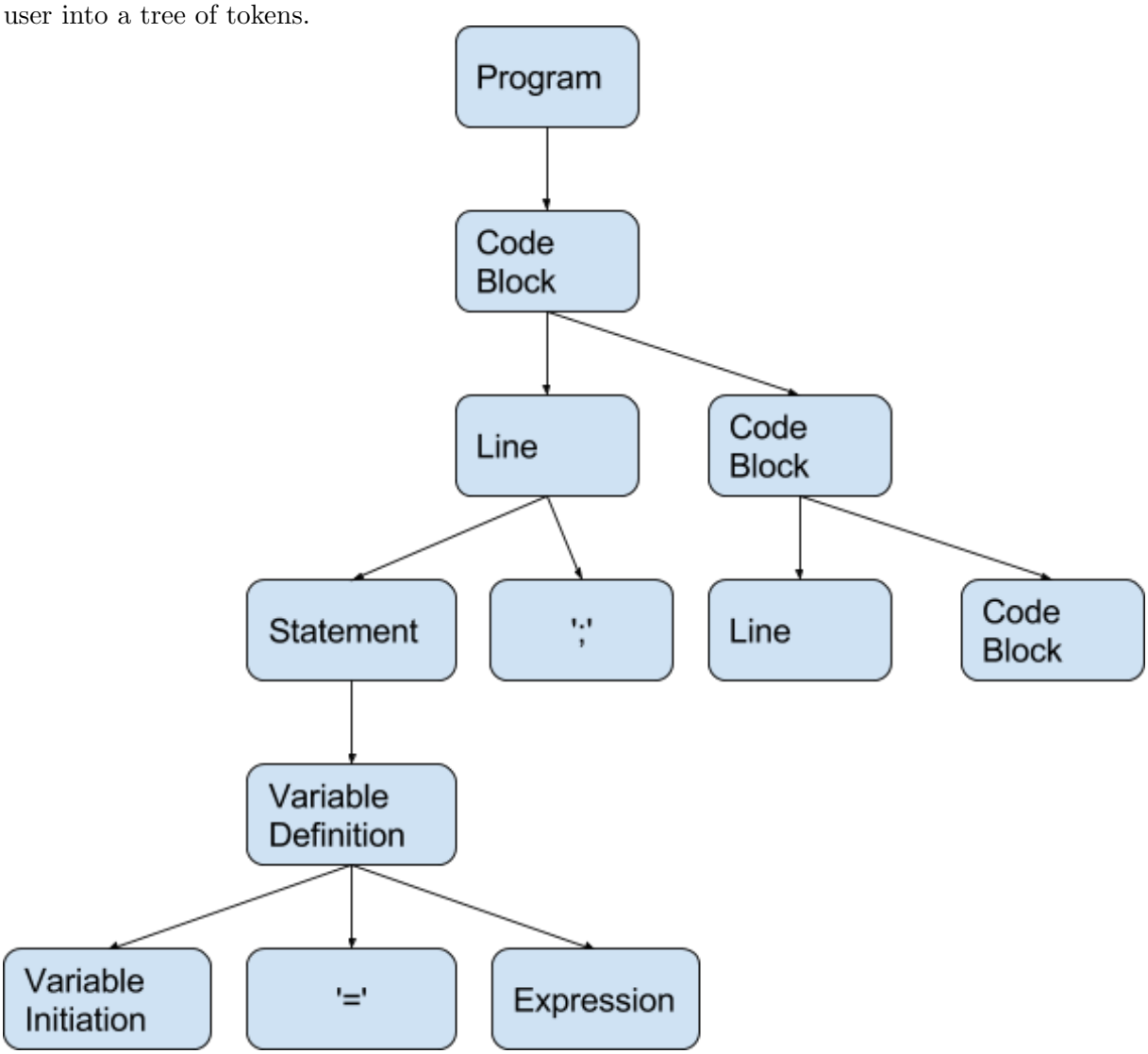
The pre processor has two main jobs.
The parser for this program takes a single string for input into which it creates a tree of tokens for the Object Generator to use. This means that the pre processor needs to concatenate the other files text onto the main file so that it can all be processed together.
This causes a second problem that the pre processor fixes. The user needs meaningful line numbers for where errors they have made occur. But with all the source files placed into one large string, the pre processor needs to know where one file starts and another ends.
For this reason, the pre processor has a list of references to where each file is represented in the large source code string. This is updated for each time a new file is added.
Another list holds the index of each line ending in the source code.

When the program needs to reference a particular point in the user's source code because an error was made, the program can convert from the index in the large source code string to a line reference from a particular file.
To do this it checks against the line ending list to see which line of the source code is being referenced. After this, if then checks the line number against the list of file references to see which file this line belongs to.
Finally to find the real line number relative to the file rather than to the source code as a whole, the number of lines that came before this file are taken away from the line number currently found.



5.2 Recursive Descent Parser

The parser uses the Recursive Decent Parser algorithm[2] to convert the code given by the user into a tree of tokens.



This token tree is generated by giving each token it's own method in the program's parser. The parser starts by calling the base token method (in this case 'Program'). The method will check to see if the user's program satisfies the requirements for this token (a single CodeBlock is a valid form for the Program token). The requirements are checked by calling the CodeBlock token's method. This will do the same as the Program's method, checking that the program satisifies the requirements for this token.

Eventually a token will be reached that has only Terminal Tokens (tokens that represent actual characters in the program rather than patterns of other tokens). Once these are reached, the algorithm can work it's way back up the tree and check to see if the rest of the program is valid in a similar manner.
If an invalid token is found the program records the point in the code in which the token failed and then tries a different possible path for the previous token. If the user's source code is invalid, the program eventually exhausts all possible forms that the program could take and fails to produce the tree of tokens. If this failure takes place the user can be notified of the point in their source code in which they have made a syntax mistake because the program always records the point in the source code that it first became aware of the mistake.

5.3 Variable and Value Storage: The VariableManager and ValueManager

The program uses a class called VariableManager to hold variables that the user has created in their code. The VariableManager class is used both to hold global program variables as well as the Variables held in objects (the 'startDate' variable is held in a VariableManager object inside the Entity object)

VariableManager

- integer num = 11
- string lit = "Hello World"
- Entity bob = Entity (In the c++ code this is held as an Entity object. This Entity object has it's own VariableManager object containing the following variables)
  - integer startDate = 1980
  - integer endDate = 2070
  - string name = "Bob"

The variables 'num' and 'lit' can be accessed anywhere within the user's program by their simple names 'num' and 'lit' respectivly.
The variables in the bob objects' VariableManager can also be accessed from anywhere (There are no private variables in this language), however the object must be noted in accessing them using the dot operator (e.g. "bob.startDate" will get the variable "startDate" in the bob object's VariableManager).

In some situations it becomes necessary for a part of the code to be able to reference two VariableManagers at the same time.
This happens for example when a method is called. A VariableManager exists for the object the method belongs to and the parameters that the method was called with. These cannot be put into the same manager as it is important for the method parameter variables to be removed after the method is finished. In order to fix this problem, the VariableManager object has a stack of pointers to other VariableManagers it needs to also look for variables in should the user's source code require it. Only the VariableManager at the top of the stack gets accessed as the mangers lower on the stack may handle method calls from further back in the code.

Some variables in the VariableManager may point to the same value that a different variable points to (This may occur when an object needs to be referenced).
Because of this, the values that these variables point to are all stored in a separate c++ class, the ValueManager.
By keeping all the values together it makes it much easier to delete value objects after the program has finished avoiding possible memory leak problems in the c++ code.
This class also allows for switching dynamic values generated by the GetEventContaining-DataValue class to be enabled for use.
Dynamic values cannot be accessed during the general execution of the source code and instead only become available when the timeline is being simulated

5.4 Dynamic Values

The GetEventContainingDataValue function that is in-built to this language requires a special kind of data type. This is because the function returns a pointer to an Event, but has not yet decided which Event it's supposed to be.
When the the function is used, it takes three parameters.
The type of event that has to be returned i.e. the class name.
A variable inside the class that will be compared against.
And the value of which to compare this variable to.
So in the following code:

```
1         MurderEvent beccaMurder = new MurderEvent(jenny, becca,
2                                         1925, "Becca's murder by jenny")
3                                         [jenny, becca];
4
5         MurderEvent beccaMurder2 = new MurderEvent(ellie, becca,
6                                         1940, "Becca's murder by ellie")
```

```
7                                           [ellie , becca];
8
9           MurderEvent originalMurder =
10                      GetEventContainingDataValue(
11                              "MurderEvent",
12                              "victim",
13                              becca);
14
15          MurderEvent revengeMurder = new MurderEvent(clare ,
16                                          originalMurder.murderer ,
17                                          1945,
18                                          "The revenge")
19                                          [clare ,
20                                            originalMurder.murderer];
```

It is known while processing line 9 of the code that the Event that originalMurder is supposed to point to is a 'MurderEvent' object, and that this object needs to have it 'victim' variable to be pointing at the object becca.

The problem at this point is that becca is the victim of two murders (beccaMurder and beccaMurder2). At the current stage of execution the program is unable to determined which murder is the one that takes place so instead of a dynamic value is used.

Dynamic values allow for the program to continue running with the expectation that any missing data will be filled in later during the timeline simulation phase. This means that much of the originalMurder object can still be accessed despite not knowing what information is truly being used, as it has been in line 16 and 20.

In the C++ code this has been implemented by inheriting from the ReturnableValue class (the class that variables hold as their value). This allows for it to be assigned to any ordinary variable. When dealing with dynamic values the C++ code makes checks to see if it should just leave the value as is with just the information on how to find the actual value, or to actually retrieve the real value.

The real value is only ever retrieved during the timeline simulation phase of the program as at this point the the program it is known which Events happen in the story and which ones don't (Dynamic values can't return values from Events that don't happen in the story)

6 Testing

A set of 29 tests have been produced in order to test the functionality of the compiler at multiple stages of the of the execution process.
These tests were split into 3 stages in accordance with the different components in the program.
These were parser tests, program execution tests and timeline simulation tests.
These three sections were further split into 2 of tests that should produce a positive outcome from the user's code being correct,
and produce a negative outcome (produce an error) from the user's code being incorrect.
Although the majority of the tests passed first time here is a list of the tests that required fixes to the compiler's code.
(These tests can be found in the main.cpp source file of the code. They can be executed by running "./main –tests", see README in the code/trunk directory for more details)

Test 13 This tested the return method functionality. However the failure of this test was caused by the use of an equation token. The equation failed to recognise an equation's operators when spaces were used around them. This required the Tokenizer.cpp[4] file to be fixed in order for these operators to be found. Once fixed a new failure was found. The test's method should have returned 'General Franco of Nationalist Spain'.
But instead found 'General Franco of "Nationalist Spain'.
This extra quote mark was also caused by a problem with the parser not recognising the correct start of a string literal token and therefore not purging the token of it's quote marks when required. A fix for this was also written for the Tokenizer.cpp file.

Test 18 This generated a segmentation fault. Up to this point ConstructorDefinition.cpp[**?**] had not effectively checked for is a ParameterSettingStatement existed before it attempted to call methods from it.
This causes a problem when no parameters are given for a method as in these circumstances ParameterSettingStatement is nullptr.

1. Test 19] This generated a segmentation fault. This was caused for the same reasons as the Test 18 failure.
FunctionCall.cpp[6] was fixed.

Test 20 This test produced a VariableNotFoundException but instead should have thrown a MethodCallException. During the process of this test, the program attempted to run the method's code, despite not having the correct number of parameters, and then produced the VariableNotFoundException at this point. MethodDefinition.cpp[7] was rewritten to recognise when the wrong number of parameters have been provided and now produces the MethodCallException.

Test 23 This test produces a TypeMismatchException but instead should have thrown a MethodCallException. During the process of this test, the program attempted to

run the method's code despite it being in an area of the code that required it to return a value. This should not have been possible as the method as marked as being an 'empty' method that didn't return a value.
Several files were fixed to ensure that 'empty' methods would not even attempt to run in such circumstances.

Test 24 This, at first failed at the parser stage despite being a vaild program.
The Tokenizer.cpp[4] was fixed to allow the first section of a subVariableIdentifier token to be a function call.
This was followed by a failure to recognise that a funcion was able to have sub variables. The SubVariableIdentifier.cpp[8] file was fixed for this.
This was followed by another failure, the program produces an error saying that it can't print objects despite the value that was being attempted to be printed not actually being an object. This was caused by the use of dynamic values in an area of the code that had not yet been equipped to handle them. FunctionCall.cpp[6] was fixed in order to produce the correct StructureException error for this circumstance.

Test 25 This generated a segmentation fault. This was caused by the SubVariableIdentifer.cpp[8] file not being equipped to handle dynamic values at this point. This was subsequently fixed.

Test 26 This produces a segmentation fault. This was caused by the ParameterDefinition-Statement object for the 'onEvent' abstract method being nullptr despite an attempt being made to access it. ClassDefinitionManager.cpp[9] code was fixed to ensure the ParameterDefinitionStatement object always exists.

Test 29 This produced a type mismatch error despite having the the types of the value and variable being the same.
This was caused by a failure to handle dynamic values in a particular area of the code.
Once fixed, a new error occured. The event given could not be found due to a minor error with an if statement (pointers to objects were compared rather than the objects themselves, this bypassed the operator== functions built into these objects).

7 Critical Analysis

Given the goal of creating a compiler for an object oriented programming language, I believe this task was achieved to a reasonable degree of success. Features of the language such as object dependency, the special function 'GetEventContainingDataValue' and much of object oriented features of inheritance and method overriding have been implemented just

as they were originally invisioned.
There is always room for improvement with any project and their are certainly areas of this project that I would change should I have had the chance to do so.
In the current software, the system used for detecting and displaying errors in the user's source code is not coherent in any way with the error messages being generated across many different classes. Because these error messages were written directly in the code there is quite a lot of inconsistency for each message. Some errors are the same but are worded differently. Some errors display the line of code where the mistake takes place while others do not.
Given the chance. The error system would have been separated from the main code and given it's own class that the rest of the system can reference. This would be much easier to maintain rather than looking through about 20 files to fix each message.
Another area of the error system that I would like to improve is errors produced by the software's parser. Under the current system the software simply tells the user that an error has occured in the parser and that there is a syntax error at a particular line of their code. This really needs to give the user information on the sort of mistake they may have made such as a missing semicolon or brackets being mis-matched.

There are also areas of the language' functionality that needs an overhaul in order to give the system a greater sense on consistency and as such greater level of user satisfaction while using the product.
One of the benefits of borrowing elements such as syntax or keywords from other languages is that the new language appears more familiar to the user than if brand new syntax and keywords are used. This however will also mean that they are more likely to expect certain functionality from the borrowed language to also be present in the new one. When the new language fails to act as they expect, this can become an area of frustration for the end user. Because of this it is clear that there are some elements of the language that need improvement inorder to feel more familiar and less frustrating to the user. The main areas I would like to have improved is the 'empty' keyword and object inheritance.
The 'empty' keyword in this language is used as the type identifier for methods that do not return a value. This keyword should really have been replaces with 'void' for it to fall in line with the expectations of a programmer familiar with C++ or Java. The problems with

using this word in this context is further exacerbated by the fact that the keyword 'void' also exists in this language to denote a method that does not hold any parameters.
The keyword 'empty' was supposed to be a placeholder for when I rewrote the parser to allow the 'void' keyword to refer to methods that didn't return values but unfortunatly, this was never realised and as such leaves the language seeming much less professional than it could have been.
Object inheritance is another area that falls short of expectation a regular programmer may have.
Although a system of object inheritance does exist for this language it is far from what the generally accepted standard for inheritance is. The inheritance system in this program simply inherits methods and variables from the previous class and has the ability for the user to override previous methods. It treats this new class as a completely separate data type however. So if class A inherits from class B, A cannot be placed into a B datatype despite this being a common feature of all other OO languages.

Finally the main area that I would likely do differently should I have started the project now is to not use the C++ language.
Much of my time in this project was spent fixing problems that simply wouldn't have existed if I was using a slightly higher level language such as Java or C#.
Files such as the ValueManager simply exist due to the requirement in C++ to clean up memory. This is an area that I wouldn't have to worry about in other languages and greatly increased the amount of time I spent on the project.
The ReturnableValue class also only exists because of problems in the C++ language. Unlike languages like C#, C++ handles objects as being inherently different to general datatypes like char or int. This meant that the values from users variables had to be handled completely differently to how I would have wanted them to be handled. All variables needed to point to a ReturnableValue object that could handle all the datatypes rather than directly to the value that could have been of any type.

8 Commercial Context

The main area in which this type of program could be implemented is in the development of video games that are heavily story based where the player's actions have an influence on how the story plays out.
Games such as the fallout or elder scrolls series often feature stories that depending on who the player chooses to make friends with have radically different endings.
These games are also famous for having characters that are impossible to kill if they are inexplicably linked to the story

Figure 1: Piper, a major character of Fallout 4, having a sitdown after sustaining multiple gunshot wounds

A system similar to the one I have attempted to create in this project could be used in a game to ensure that where major player characters from a story have died, the story doesn't die with them and instead continues with other characters taking the place of the fallen. Just because the king dies doesn't mean the player should be denied the story of storming the kingdom.
The system in it's current form would require some significant work in order to become useful for a major game developer, due to these developers often having a rather complex scripting system for creating stories. However a smaller game project may find such a tool quite useful as a means to create a branching narrative that they would normally avoid due to the amount of time it takes to produce.
With a traditional branching story system two stories that are immensely similar could be

required to be written out twice because the characters in the story had changed. This can be merged into a single story line with this new system, but the player would still be given the same amount of satisfaction against half as much work for the programmer.

9 Personal Development

Although mentioned earlier in my critical analysis that given a second chance on the project I would not use the language. The area in which I improved the most is my skills in writing C++ code. Using the C++ language in such a large project has forced me to learn many things about the language that I would otherwise have struggled with in my future career. Areas such as memory management and understanding how pointers work in a C++ program have improved dramatically as a result of the challenges faced in making this software system a reality.

I have also gained a great deal of experience in using recursive programming techniques. Almost every component in my program uses recursive techniques to some degree and as such using recursion in future projects will not nearly be so daunting as it would have been prior to the advent of this programming task.

10 Conclusion

The aim of this project was to create an object oriented programming language for the purposes of allowing the programmer to generate branching stories that were resilient to the typical bugs created by missing major characters or other such similar failings.

Although their are still areas in which the software can be improved. I believe that the aims of this project have been met to an acceptable standard. A functioning compiler with good error checking and a feature rich programming language has been produced that meets the expectations of the projects inception.

References

[1] Leonhaut, R.v.,
The code for the Object Generator (ObjectGenerato.h/.cpp) is based on this implementation of the GOLD Parser ,
`http://www.goldparser.org/doc/index.htm`

[2] *The code for the parser (Tokenizer.h/.cpp) is based on the C implementation found on the recursive descent parser algorithm's wikipedia page*,
`https://en.wikipedia.org/w/index.php?title=Recursive_descent_parser&oldid=740696944#C_implementation`

[3] *The backus naur code uses the gold parser's implementation of Extended Backus Naur Form*,
`http://www.goldparser.org/articles/bnf.htm`

[4] *This source file produces the tree of tokens required for the object generator to run*,
`../../code/trunk/src/Parser/Tokenizer.cpp`

[5] *This source file is used to represent constructors in the users code during object generation*,
`../../code/trunk/src/ObjectGeneration/TokenObjects/ConstructionDefinition.cpp`

[6] *This source file is used to represent function calls in the user's code during object generation*,
`../../code/trunk/src/ObjectGeneration/TokenObjects/FunctionCall.cpp`

[7] *This source file is used to represent methods in the user's code during object generation*,

../../code/trunk/src/ObjectGeneration/
TokenObjects/MethodDefinition.cpp

[8] *This source file is used to represent sub variables of objects being accessed in the user's code during object generation,*
../../code/trunk/src/ObjectGeneration/
TokenObjects/SubVariableIdentifier.cpp

[9] *This source file is used to hold information on all the existing class types in the user's code,*
../../code/trunk/src/Variables/
ClassDefinitionManager.cpp