Effects of Error Messages on a Student's Ability to

Understand and Fix Programming Errors

by

Harsha Kadekar Beejady Murthy Kadekar

A Thesis Presented in Partial Fulfillment

Of the Requirements for the Degree

Master of Science

Approved August 2017 by the

Graduate Supervisory Committee:

Sohum Sohoni, Chair

Scotty D. Craig

Shawn S. Jordan

Kevin A. Gary

ARIZONA STATE UNIVERSITY

December 2017

ABSTRACT

Assemblers and compilers provide feedback to a programmer in the form of error messages. These error messages become input to the debugging model of the programmer. For the programmer to fix an error, they should first locate the error in the program, understand what is causing that error, and finally resolve that error. Error messages play an important role in all three stages of fixing of errors. This thesis studies the effects of error messages in the context of teaching programming. Given an error message, this work investigates how it effects student's way of 1) understanding the error, and 2) fixing the error. As part of the study, three error message types were developed – Default, Link and Example, to better understand the effects of error messages. The Default type provides an assembler-centric single line error message, the Link type provides a program-centric detailed error description with a hyperlink for more information, and the Example type provides a program centric detailed error description with a relevant example. All these error message types were developed for assembly language programming. A think aloud programming exercise was conducted as part of the study to capture the student programmer's knowledge model. Different codes were developed to analyze the data collected as part of think aloud exercise. After transcribing, coding, and analyzing the data, it was found that the Link type of error message helped to fix the error in less time and with fewer steps. Among the three types, the Link type of error message also resulted in a significantly higher ratio of correct to incorrect steps taken by the programmer to fix the error.

DEDICATION

I dedicate this work to my mother *Prema Kadekar* and my father *Murthy Kadekar* who are making my dreams a reality. *Sowmya Rao*, my sister who always stood by me silently.

Dodappa (B. C. Rao) who believed in me and always found ways to make it happen. *Suma Chikkamma* my second mother who is always there. My cousin, well more of an elder brother *B. C. Mohan Rao* – you made me to dream, encouraged me to take steps to achieve those dreams.

SSAHASS – you guys always have a torch when I get lost in dark. Thanks Yash for guiding.

ACKNOWLEDGMENTS

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVATIONS

1. PLP – Progressive Learning Platform

2. IDE – Integrated Development Environment

3. FPGA – Field Programmable Gate Array

CHAPTER 1

INTRODUCTION

1.1   Motivation

Assemblers and compilers act as feedback mechanisms which helps the student programmer to test their mental models of programming (Traver, V. J., 2010). Each time they use assembler/compiler, it will evolve those mental models. Assembler and compiler error messages are typically formed from the perspective of the programmer designing the compiler or assembler, rather than the perspective of the user of these tools (Traver, V. J., 2010). Among the student programmers, novice programmers are the ones who are most affected by these error messages (Nienaltowski, M. H., Pedroni, M., & Meyer, B., 2008, March).

Usually an error message describes at what stage of assembling or compiling an error occurred rather than describing what could have caused the error, or what mistake on the programmer's side could have caused the error. For a student programmer who does not have any background on the inner workings of the assemblers/compilers, these messages appear to be cryptic and unhelpful (Traver, V. J., 2010). Thus, usually the novice programmer ends up making random changes, following an unguided trial and error process hoping to get rid of the error. When the cryptic error messages persist, or increase during these attempts to fix the error, the programmer's frustration increases (Rodrigo, M. M. T., & Baker, R. S., 2009, August). So, as many novice programmers treat these errors as personal failures, they feel demotivated, and their fear of programming increases (Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L., & Zander, C. ,2008).

There is a direct relation between error messages and debugging. There are three steps in the process of fixing the error. First would be to identify the location of the error in the program. The next step is to understand the cause of the error based on the error message provided. The final step is to fix the error based on the information provided by the error message. Basically, the error message generated by the assembler/compiler should help the programmer in locating, understanding, and finally fixing the bug in the code (Lazonder, A. W., & van der Meij, H., 1995). With cryptic error messages, it takes more time to understand why the error is occurring. Programmers who do not understand the error message will either misinterpret the error or completely lack the understanding of what is going on in the program, and move towards trial and error. Thus, it would take longer to fix the error. This leads to an overall increase in programming time and programming steps as debugging will take a longer time.

If a significant part of the course involves programming, the frustration caused due to cryptic and unhelpful error messages negatively affect the motivation, engagement and learning in the course (Rodrigo, M. M. T., & Baker, R. S., 2009, August). As error messages are the direct feedback for the students' learning models, cryptic error message will become a barrier in the development of those learning models. However, useful error messages will not only help in correcting the error, but will also help to build a strong mental model of the programming language in the student's mind. They will help clarify concepts, remove misconceptions, and generally result in better outcomes for programming intense courses (Traver, V. J., 2010).

This thesis explores the impact of different types of error messages on a student's understanding of the error as well as fixing of the error. The study was conducted using assembly language programs and their syntax errors.

1.2   Review of Literature

There are multiple studies conducted over many decades to understand the effects of the error messages and improving those error messages to help programmers. The studies vary from identifying most common types of errors committed by the programmers, understanding compiler error messages in the view of human computer interactions, how the programmers debugging process varies, comparing the novice and skilled programmer's ability to handle error messages, understanding the error message wording effects on the programmers, providing a better representation for the error messages and so on.

In the study conducted by Chabert J.M. & Higginbotham T.F., 1976, the errors committed by the novice programmers were tabulated based on the type of errors and frequency of errors committed. Along with thisrecommendations were provided to improve the course work and update the assembler design so that it facilitate the reduction of the number of errors by novice programmers. Here target programming language was IBM 370 (OS) assembly language. In our study also, we are using assembly language and one of the future goals of our study is to explore possibilities to update course work and language design.

The study by Marceau, G., Fisler, K. & Krishnamurthi, S., 2011, helps to quantify the effectiveness of the error messages for a novice programmer.. This study explains that an error message is effective if a student reads it, can understand its meaning, and then use the information to formulate a useful course of action. Based on this theory, various codes were developed to analyze how students programmed given an error message. The programming activity was coded and this data was used to develop a rubric. The rubric was used to develop a formula which helps to quantify the students action on the program based on the presented error message. This study like ours, uses

human-factors research methods to explore the effectives of error message. "Read->Understand->Formulate" theory is also used in our study to understand the students programming activity.

Vessey I., 1985 used verbal protocol analysis to understand the debug process employed by an expert programmer and a novice programmer. The study found that even though both expert and novice used breadth first approaches for debugging but the novice is deficient in their ability to think in system terms. The novice is also poor in chunking the programs when compared to expert programmers. This was the first study which employed Think aloud experiment to capture the mental model of a programmer. In our study, also we employ think aloud programming activity to better understand the mental model of student programmers.

Nienaltowski M.H., Pedroni M. & Meyer B., 2008 studied three types of error message representations – short form, visual form, and long form. Students from two different universities took part in the study. The study tested hypotheses like "higher experience results in faster answers, higher experience results in more correct answers, at a lower experience level enhanced messages result in more correct answers, more information results in more correct answers, the error type determines number of correct answers, more information in the error messages results in shorter response time". Two set of multiple choice questionnaire was prepared. The questionnaire had 3 types of errors in 3 different forms of error messages. The results showed that providing more information in the error message does not lead to more correct answers. It also showed that giving more information in the error message did not reduce the response time. The other hypothesis which proved wrong was type of error determines the number of correct answers. In our study, also we have one short form error message and two long form error messages. In the long form error message, we are providing more

information and description is enhanced when compared to short form error message. Our study gave different results as enhanced error messages took less time to resolve error and less steps to resolve it.

In the study conducted by McLaren B.M., DeLeeuw K.E. & Mayer R.E., 2011, given the error message in a polite language, the student with low prior knowledge of the subject performed better when compared to students who were exposed to error message which was in direct language. A web based intelligent tutor was created which students used to solve chemistry stoichiometry problems. The tutor provided hints and feedback in either polite or direct language. In our study, two of the three error message types, provides the error description in detail. They are well organized. The findings of our study were like that of study conducted by McLaren B.M., DeLeeuw K.E. & Mayer R.E. 2011.

Lazonder A.W. & van der Meij H. 1995, studied the effects of error information present in a (tutorial) manual. The study revealed that subjects who are exposed to manuals that have more information on the error performed better when compared to the subjects who were exposed to manual which are having less error related information. It explains in detail about three steps of error correction – detection, diagnosis, and correction. A think aloud activity was conducted in the study. As part of the study two groups of 25 participants took 3 types of tests – constructive sill test, corrective knowledge test, corrective still test. One group was provided with manual having more information about the error message and another group was provided with manual having less information about the errors. Each test measured different aspects of error detection, diagnosis, and correction. In our study, one of the error message type will be providing hyperlinks to relevant sections of the online manual to help the

participants in diagnosing the error and then resolving the error. The error message type which had the links showed better results when compared to other error message types.

Hartmann, B., MacDougall, D., Brandt, J. & Klemmer, S.R., 2010, April developed a system which gives helpful suggestions to compiler and runtime error messages. This system called as HelpMeOut, tracks the source code development from its error state to final correct state. It stores all the compiler errors committed by the users in a central database – what was the error message, what was the wrong code and how that was corrected. When a user asks for help to correct an error message, it will fetch relevant data from that central database. Suggestion would include a description of possible fix, explanation of the error and previous code examples of error and how it was fixed. This was done for 2 programming languages – Java and C++. A total of 39 hours of programming data was collected out of which 178 times suggestions were provided. Among them 47% of them were useful. In our study, one of the error message type has similar feature of displaying a relevant example code.

Traver, V.J., 2010 studies the problem of cryptic compiler error messages in the perspective of human-computer interaction to understand why error messages make the work of programmers more difficult. This study talks about how current error messages are more of compiler centric rather than programmer centric. When the error messages were analyzed from the human computer interaction perspective, it was observed that most of the error message were lacking in clarity and, they were context insensitive. Most of the error messages were lacking in constructive guidance as well that means most of the error messages were not polite and were not providing valid suggestions for correcting errors. It also observed that many of the error messages were not specific which resulted in different diagnostics. In our study, error messages were designed to rectify the above mentioned lacking points.

6

1.3   Problem Statement

This study tries to understand the effect of an error message on different aspect of resolving an error. This is an exploratory study, which tries to understand mental model of a programmer and how an error message affects that mental model. As part of the error message construction itself, study tries to find out what error information are more helpful to programmers. It tries to find out if an enhanced and more detailed error message help in improving the performance of programmer when compared to short cryptic error message. To better capture all the different features which needs to be tested as part of the study, three types of error message types were developed.

1. Default – Here error messages will be short and messages indicate at which stage of assembling the error occurred.
2. Link – This will have detailed error messages and a weblink address for the section of online manual where more information can be obtained
3. Example – This will have detailed error messages and sample code. The sample code has two parts one with similar error as currently encountered by the participant and other one with the fix for that sample code error.

1.4   Research Question

By analyzing the programmer's interaction with respect to three types of error messages and their corresponding programs, this study tries to answer following two research questions.

1. What aspects of an error message help the programmer to understand the error?
2. What aspects of an error message help the programmer to fix the error?

CHAPTER 2

IMPLEMENTATION

The programming language used for the study is an assembly language called Progressive Learning Platform (PLP) and the tool used to assemble the program is PLPTool. "Progressive learning platform is an FPGA based computer architecture learning platform, and was designed for students to anchor their conceptual learning about microprocessors and computer architecture, and for them to see the connections between assembly language and trade-offs in architecture" (Sohoni, S., 2014, June).

PLPTool and the assembler present in that tool was modified as part of this study. A separate module for processing the assembler exceptions were created. Once the assemblers discover that an error exists in the code, instead of raising the exception, this module will be called. The control will be passed to this module along with the program instruction which failed to parse, as well as other information needed. Then in the module, a detailed analysis is conducted on the error.

Overall assembling process of the PLP language was studied and based on its working and PLP language structure, each possible error was grouped into 4 types.

1. Invalid label – This error has two sub groups.
    a. Duplicate label – This error occurs when the same label is used in two different contexts.
    b. Invalid target – This error occurs when the program is using a label name which is not yet defined.
2. Invalid token – This has two sub groups.
    a. Invalid instruction type – This error occurs when there is a spelling mistake in the instruction keyword.

8

b. Invalid label – This error occurs when the label is not declared like missing a colon.

3. Invalid number of tokens – This error is caused due to an invalid number of arguments to the instruction. It has two sub groups

    a. Missing tokens – This error occurs if there are less tokens then expected

    b. Extra tokens – This error occurs if there are more tokens then expected

4. Invalid operand – This error is caused due to invalid operand to the instruction. It has four sub groups.

    a. Not Register – Instruction was expecting a register but got something else.

    b. Not Number – Instruction was expecting a number but got something else

    c. Not String – Instruction was expecting a string but got something else

    d. Invalid address – Instruction was expecting an address value but got something else.

Once this categorization of errors was completed, I created an error information repository. For each type of error following information was stored in the repository.

1. Description – This gives a detailed explanation of the error. Here explanation would be program centric rather than assembler centric.

2. Links – This gives an http link to a section of the online PLP manual, where the programmer can get more useful information for understanding and fixing of the error.

3. Examples – This gives a code sample. It has two sub sections - before correction and after correction. Before correction has an example instruction which has the error. After correction has the fix for the same error sample code.
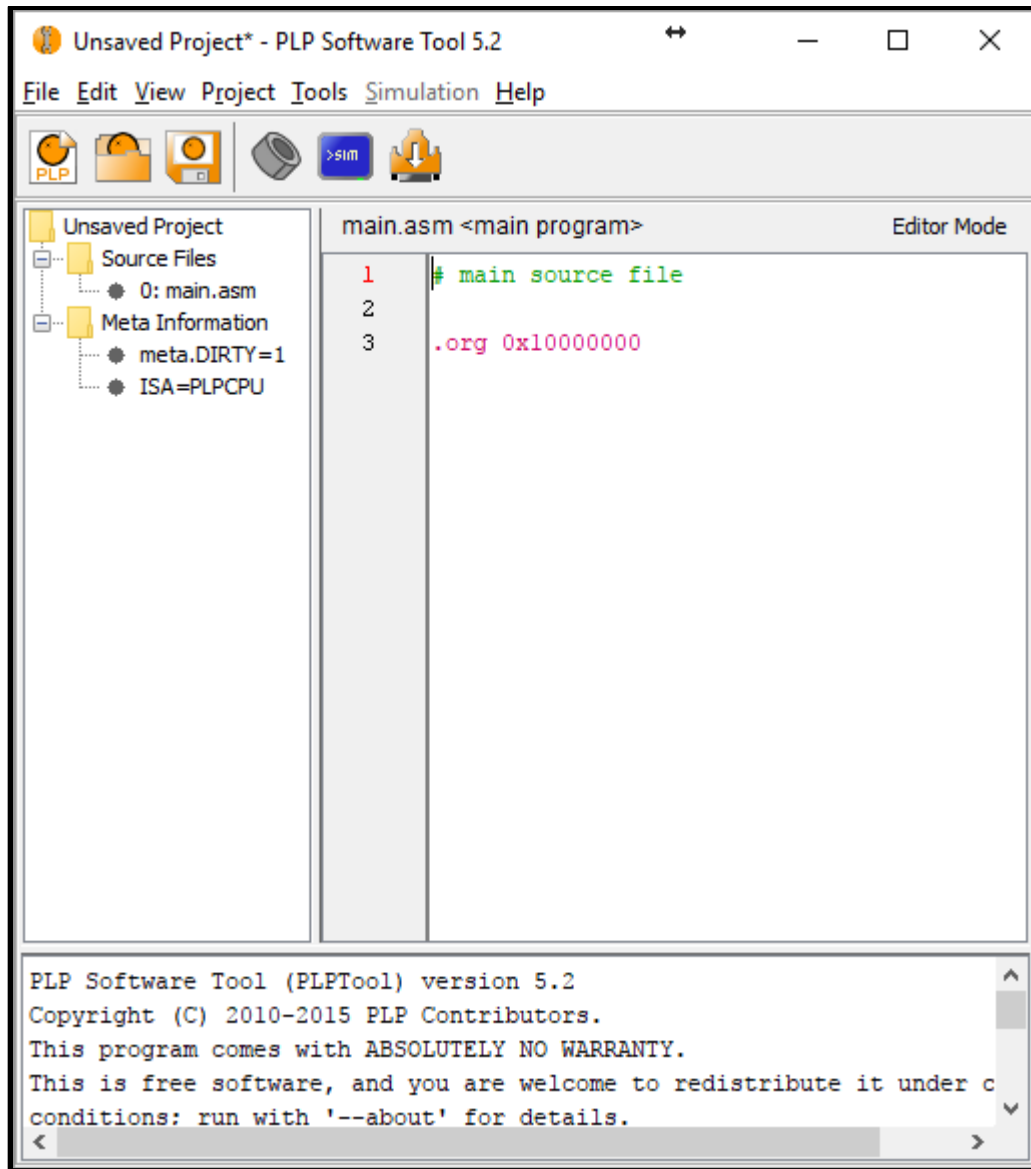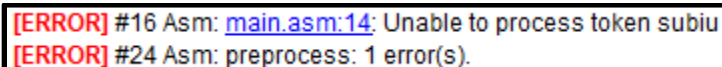
Figure 1- PLPTool

Once the type of error is identified, error information will be retrieved from this repository. Based on the type of error message, some information will be discarded, and the rest will be formatted and displayed in the console window of the PLPTool. If the

Link type of error message is to be displayed, then example code part will be discarded. If the *Example* type of error message is to be displayed, then link will be discarded. If the *Default* type is chosen, then all the above information will be discarded and only an assembler-centric, single sentence error message will be provided.

Every error message will have an error number and error location link which upon on clicking will highlight the line where the error occurred. The link will have the line number and file name where an error has occurred.

The idea of error message with code examples was adopted from Hartmann B., MacDougall D., Brandt J. and Klemmer S.R 2010 [9], where they had developed a centralized repository which collects all the errors committed by the programmers and the corresponding fixes done to those errors by the same programmers. Any time a programmer commits an error, the error message will have the previous corrected code samples thus turning the whole exercise into a community driven one.


```
[ERROR] #16 Asm: main.asm:14: Unable to process token subiu
[ERROR] #24 Asm: preprocess: 1 error(s).
```

Figure 2- The *Default* Type of Error Message

Figure 2 provides the *Default* type of error message. Instruction which is causing the error is subiu $s1, $t3, 10. For a novice programmer, who does not know the inner working of assembler or compiler, they will not know what is "token." Error also says, "unable to *process* token "subiu." So, novice programmer might infer that the way instruction and its operands is written might be wrong and hence processing of that instruction failed. The actual reason for the error is PLP does not have any instruction by name "subiu." Instead in PLP, subtract operation can be performed by subu instruction.

11

Figure 3 provides the *Link* type of error message and Figure 4 provides the *Example* type of error message.


```
[ERROR] #16 Asm: main.asm:14 This error is caused due to wrong instruction name. PLP
does not have an instruction as mentioned in the code.subiu - instruction is not defined in
PLP. Unable to process the instruction.Error occurred around word - "subiu"
Please refer following link for more information http://progressive-learning-platform.github.
io/instructions.html#operations
[ERROR] #24 Asm: preprocess: 1 error(s).
```

Figure 3- The *Link* Type of Error Message


```
[ERROR] #16 Asm: main.asm:14 This error is caused due to wrong instruction name.
PLP does not have an instruction as mentioned in the code.subiu - instruction is not
defined in PLP. Unable to process the instruction.Error occurred around word - "subiu"

BEFORE CORRECTION
sub $t1, $t2, $t3
AFTER CORRECTION
subu $t1, $t2, $t3
[ERROR] #24 Asm: preprocess: 1 error(s).
```

Figure 4- The *Example* Type of Error Message

To conclude that it's an invalid instruction, we need to first check how many tokens the line has. If one or more tokens, follow the unidentified token than it is a case of wrong instruction name that is an error type INVALID TOKEN with the sub group as INVALID INSTRUCTION TYPE. If the line has only one token which is unidentified, then this might be the case of a missing colon for a label that is error type would be INVALID TOKEN with sub group as INVALID LABEL. Once we know the error type and its sub group we can query the error information repository to get more information like detailed description, links, and example code. There are few instructions which have only one word and an error can occur if that word has a spelling mistake. As of now it will be flagged as an invalid label declaration error. In future, the error analysis had to be improved further to handle such scenarios as well.

CHAPTER 3

METHODS

3.1 Design

To examine the effects of different types of error messages on the student's ability
to understand and fix the errors in the program, we need to first understand the
knowledge model of the student who is going to fix the error. To capture this knowledge
model effectively, think aloud programming was used. The experiment had three
components.
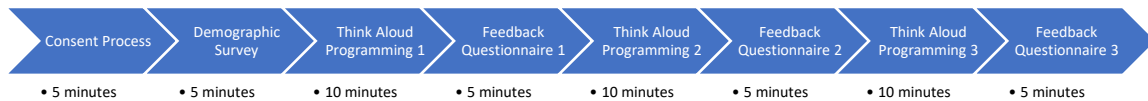
Figure 5- Experiment Stages

1. Demographic Survey - This survey is done via Qualtrics online survey tool after
   the subject gives electronic consent. This survey is conducted to know more about
   the participants programming background

2. Think Aloud Programming Activity - Think aloud activity helps in "obtaining a
   real-time insight into the knowledge that a subject use and the mental process
   applied while performing a process of interest" (Hughes, J., & Parkes, S., 2003).
   Think aloud represents the subjects working memory. The terms verbal reports,
   verbal protocols, think aloud protocol and talk aloud protocol are used
   interchangeably, there are very few differences between each of these protocols.
   "It refers to human subject's verbalizations of their thoughts and successive
   behaviors while they are performing cognitive tasks" (Ericsson, K. A., & Simon,

H. A. 1993). In this study, for each participant think aloud programming exercise
was done for 3 PLP programs.

3. Feedback Questionnaire - Questionnaires had questions related to role of error
message in programming activity. These questions were aimed for collecting
views of what helped and what did not help in the given error message type.

The whole study took approximately one hour to complete for one participant. Figure
4 represents the different steps of the experiment.

3.2   Procedure

Demographic Survey – As part of this survey, questions were asked to understand the
proficiency of participants in the subjects like PLP, assembly language, high level
programming language and usage of Integrated Development Environment(IDE). Each
question was given three answer choices - novice, intermediate and expert. It would be
better to choose a 4-scale option rather than three as participants tend to choose the
middle option. We can avoid this scenario by using a number scale like 2, 4, 6 and 8 with
2 being least proficient and 8 being expert. 5 minutes was given to complete the survey.

Think Aloud Programming Activity –In this study we are trying to understand the
knowledge structures student programmers use to solve the error and how those
knowledge structures are varying due to the error message, which was the input given to
the students. The study is more interested in the process followed by the participants to
solve the error rather than final result of whether error was fixed or not. As Vessey I.,
1985 mentions, verbal protocol or think aloud protocol is the preferred method of
examining problem-solving process. Think aloud protocol help us to capture more data
and captures how different features of input was used by the participants. In multiple
research (Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas,

14

L., & Zander, C. (2008); Letovsky, S. (1987); Jeffries, R. (1982, March)) think aloud protocol is used to understand the mental model of the programmers

In this study, think aloud programming exercise was done for 3 PLP programs. Each program had a single error. Participants were asked to correct the error. For each program, 10 minutes time was given for the participant to understand the error and fix it. PLP program was opened in PLPTool, the development tool used in this study. After 10 minutes, an online feedback questionnaire was provided. This process was repeated for other two PLP programs.

Before the start of the think aloud programming exercise, following instruction was provided to the participant - "*I will give you a PLP program. Program has one single error. Please keep talking aloud while correcting the error in program. You have 10 minutes time. It is okay, if you will not be able to fix the issue within 10 minutes. Please try to say everything that goes through your mind.*" When participant stops talking and there is a silence for more than 15-20 seconds, then subjects were prompted by showing a placard with sentence "*Keep on talking*" (Van Someren, M. W., Barnard, Y. F., & Sandberg, J. A. C. , 1994). The complete session was screen recorded, i.e. whatever happened on the computer screen during that session was recorded, and the session's audio was also recorded. Thus, for each participant, a maximum of 10 minutes of screen and audio recording was done per program, for three programs.

Feedback Questionnaire – After each programming exercise completion, a feedback questionnaire was provided to participants. Questionnaires had questions related to role of error message in programming activity. Following questions were asked

1. Could you explain in your own words what was the error in the program?
2. How did the error message help you to understand the error?

3. How did the error message help you to fix the error?

These three questions were asked after completion of one PLP program. Participants were given 5 minutes minimum to complete this questionnaire, but they were free to take more than 5 minutes to fill out the questionnaire. The questionnaire was provided as an online form. The main reason for questionnaire was to capture those points from the subjects which were not verbalized during the programming activity itself. This provides another medium for the participants to express what helped to build their thought process. This is useful for the participants who generally not used to talk.

3.3   Materials

For each participant, 3 programs were provided to solve. Each program had one error. Please refer the Appendix A for the actual programs presented to the participants. Each program had the program description in comments at the beginning of the program. The program description explained what is the intent of the program or what is that program trying to do. Following are the three programs provided to participant. Each participant faced these programs in the same order.

1. Label Program: This program had invalid label declaration error
2. Instruction Program: This program had invalid instruction error
3. Register Program: This program had invalid register error

There are 3 types of error messages representation which are tested –

1. Default Type: Existing error message type which has single line error description which is assembler-centric.
2. Link Type: Descriptive program-centric error description with hyperlinks to get more information

3. Example Type: Descriptive program-centric error message with code examples related to error.

Please refer Appendix A for detailed information about each program, different types of error messages displayed and correct fix for each of the errors. These error types were chosen based on observations we had in SER 250 classes. Based on discussion with other TA's, it was decided that above types of assembler errors were most common among the students. A detailed study on the type of errors and the time taken to fix them by the novice students' needs to be done in future Table 1 provides the treatment groups or the order by which type of error messages were exposed to each of the subjects.

	Label Program	Instruction Program	Register Program
Subject 1	Default	Link	Example
Subject 2	Link	Example	Default
Subject 3	Example	Default	Link

Table 1- Treatment Groups

3.4 Participants

PLPTool is used in Advanced Computer Architecture (SER 520) and Microcomputer Architecture and Programming (SER 250) courses in Arizona State University Software Engineering program. For this study, any students who are currently enrolled in SER 250 or SER 520 subjects or previously had taken SER 250 or SER 520 were eligible to take part in the study. SER 250 is an undergraduate level course and SER 520 is a graduate level course. A short document, detailing the research work and actual work involved in the experiment was developed. This document was sent to all students who were taking SER 250 and to the students who are part of the Software Engineering

program via email. Any student who met the pre- requisites and interested to take part in the experiment contacted via email address given in the document. Please refer APPENDIX C for the consent/recruitment document. Those who volunteered for the study and completed it were given $10 worth of gift card. 12 participants took part in the study. To protect the identity of each of the participants, they were assigned a random number between 1 to 100. All the data related to that subject were stored and referred using that random number. Among the 12 participants, 6 participants were undergraduate students who were taking SER 250 at that time or taken the course previously. Remaining 6 participants were graduate students who had taken SER 520 previously. Subjects should be familiar with PLP language and PLPTool before taking up the study as they will be spending time using PLP language and PLPTool during the study. This is the reason for setting the prerequisite of SER 250 or SER 520 course for taking part in the study. As per Hughes J. & Parkes S. 2003, who reviewed the techniques of verbal protocol analysis used in software engineering research, "the number of subjects utilized in software engineering studies tends to be small, with fewer than 30 protocols being collected. Usually a sub-set of these are only used in the next stages of the research, often with 10 or fewer being prepared for encoding… Experiments normally take between 1 and 2 h to be completed, although shorter recordings have been reported" (Hughes J. & Parkes S. 2003). So, this experiment with participant pool of 12 falls within the previous research standards.

3.5 Transcribe, Segment and Code verbal data

All the participants recording were transcribed. Transcription involved both verbal utterances as well as screen activity like changes in the program, searching in the online manual. Transcribing was done in multiple iterations. In the first pass, complete screen recording of that program was viewed along with the audio. In the second pass,

just the audio of the recording was transcribed. Finally, in the third pass, the corresponding screen activities were transcribed and linked with the transcript of the audio recording. If any words were not audible in the recording, they are transcribed as [inaudible words]. If there are a few seconds of silence, then it is transcribed as '...'. Similarly, if there is a long pause, it is mentioned in the transcript as [long pause]. Please refer to Appendix B for a sample transcription and coding of data.

Steps as prescribed by Chi, M. T., 1997, were followed for coding verbal data. Those steps are

1. Reducing or sampling the protocols

2. Segmenting the reduced protocols

3. Developing a coding scheme

4. Operationalizing evidence in the coded protocols that constitute a mapping to some chosen formalism

5. Seeking pattern(s) in the mapped formalism

6. Interpreting the pattern(s)

7. Repeating the whole process, perhaps coding at a different grain size.

Reducing or sampling the protocols – Even though screen and audio was recorded for the whole session, transcribing of the data was done for parts where the participants solved the program.

Segmenting the reduced protocols – Segmentation was done based on the activity. Here activity can be reading a program code, assembling the code, making changes to the code, searching for information, evaluating the code changes or hypothesizing a solution. Each segment can consist of just verbal words spoken by the participants or just the screen changes done by the participants or combination of both.

19

Developing a coding scheme – It took 4 iterations to come up with the codes necessary to analyze the data. The first set of codes define the activity which the participant is doing. This forms the basic steps. Each step or segment forms one activity. Table 2 gives the codes and their definitions of it.

When participant is trying to go through the information presented in front of him/her, that step will be treated as examine. Each examine step is further coded to understand what exact activity is being performed by the participant. These extra codes which define the Examine activity are given in Table 3.

A basic step can be further coded as correct or incorrect step. An expected/correct step is a step taken by the participant which will help to resolve the error in the program. This usually means participant has understood the error and is in the right direction to fix the error. Any step which is not helping to understand or fix the error can be classified as an unwanted/incorrect step. Among our basic steps, Examine and Evaluate cannot be classified like this but the rest, (Explore, Hypothesis and Repair steps) can be further coded as Expected or Unwanted. There are different possibilities for the unwanted steps. Table 4 gives the list of codes which can be used to define the Explore, Hypothesize and Repair step.

Error in Instruction program is that an invalid instruction subiu is used to do the subtraction. So, the right step would be to use subu instead of subiu or use addiu with negative 10 to do the operation. If a participant's action reflects any of the above two steps that will be treated as Expected. For example, "*so I think that means that it supposed to be subu I think it is I remember have seen this problem before*" – this is an expected hypothesis.

Gaming occurs when a participant tries to copy the contents of the error message to fix the code without understanding what is the error or how this is going to fix the error. Participant faced an error message as "[ERROR] #68 Asm: main.asm:19 Register used is not recognized by the PLP. Addu is expecting a register but got something else 10. Error occurred around word – "10" ". To fix this error, participant replaced 10 in subu $s1, $t3, 10 instruction to "10" as shown in the error message resulting in instruction subu $s1, $t3, "10". Now this is an example of gaming as participant has not understood what is the error and how the fix is going to help.

Code	Definition	Examples
Examine	Action involving reading program description, program code, error message and reading searched information	"addition on v0 subtraction on v1 multiplication on v2"
Explore	Actions involving searching in internet, PLP Online manual, PLP Quick reference guide and error location link	"Goes to online manual. In that opens Register names and conventions section."
Hypothesize	A possible solution or an explanation given by the participant	"hmmm… so addiu is for sign extended addition but there is no I am not able to find any command for subtraction so I think we should use subu"
Repair	Doing the code changes in the program	"Line 14 is changed. Replaces subiu with subu. Now line 14 has instruction subu $s1, $t3, 10"
Evaluate	An action taken to test	Assembles the

the repair or hypothesize.	program by clicking assemble button. Gets one error. [ERROR] #68 Asm: main.asm: 14 Invalid Register(s)

Table 2- Basic Steps Code

Code	Definition	Examples
Program Description	Examining the given description of the program	"this program performs addition subtraction and multiplication operation on two registers $a0 and $a1"
Code	Examining the code of the program	"So, the first the assembler direction org is done then we are loading ahh t1 with 0 t2 with 45 t3 with 60"
Error message	Examining the error message displayed in the tool	"hmm so now it is saying invalid registers line 14"
Search Information	Examining the information which was searched either	"jump register instruction can be used to this return address

	through internet, PLP Online Manual or PLP Quick Reference	jump back to this thing load the content of ra…"

Table 3- Examine Step Codes

Code	Definition
Expected	Action taken by the participant was expected by us
Gaming	Error was fixed using the error message but participant did not understand the error or the solution
T&E	Trial and Error
Correct Independent	A correct action was taken independent of the error message
Incorrect Interpretation	Incorrect action was taken based on the error message
Incorrect Independent	An incorrect action was taken independent of the error message
Silly Mistake	Interpretation was correct but made a mistake while fixing it

Table 4- Correct and Incorrect Steps Code

Trial and Error occurs when a participant does a code change(repair) followed by assembling the changed code. On finding an error, they revert the changes done. So usually it will have Repair – Evaluate – Repair steps.

When participants tend to do wrong code fixes or come up with wrong solutions based on the read error message, it shows that participant has incorrectly interpreted the error message. "*After assembled I am getting error at main asm 20 unable to process token addition… line 20 that would be ahmm it says addition on that line and it appears it could be either a comment*" – Here participant interpreted the error as possible missing '#' for making that line as comment but it is missing ':' to make it as a label.

If the code changes done or hypothesis are not related to the displayed error message and the code changes done is not correct, then that is categorized as Incorrect Independent - "*Okay now I am getting error invalid register…Okay I got the error actually we are checking branch on equal t1 with $0 it should be 0 only*". Here error occurred due to usage of an integer 10 instead of register in instruction *subu $s1, $t3, 10* and the error was at line number 14. The participant thinks that error occurred at line 12 in instruction *beq $t1, $0, increment*. There is no relation between the error message displayed and the solution came up by the participant.

When a participant comes with the correct solution for the error without understanding what is the error or without taking the help of error message, then correct independent occurs. This scenario usually occurs when participant tries different ways to fix the error unsuccessfully and suddenly does the correct change without giving any explanation of why this change is done or does not explain what is the error. This is little difficult to identify as some participant do not express it while solving the problem. This

could be a case where participant has recalled a previous experience which helped them to correct the error. Usually I look for their answers in the feedback questionnaire and as well as pattern of how the corrections were done until that step to decide whether correct independent has occurred.

When participant clearly understands what is the error but while fixing it does some syntax error, this is treated as Silly Mistake. "*Changes done in line 28. $v2 is replaced with again $v2. So final change is same as the prior to edit mul10 $v2, $a1, $a0*". In this case participant knows that $v2 is a wrong register but while fixing it again places $v2 instead of some other register thus causing an error.

Codes were also generated to analyze whether participants read the error message complete, partial, or ignored it. Table 5 gives 3 codes used for identifying how much error message was read by a participant.

Code	Definition
Complete	Participant reads error message completely
Partial	Participant reads error message partially. Usually they only read the description and ignore the link provided or sample code provided with the error message
Ignore	Participant did not read the error message. Participant solves an error without even reading the error message.

Table 5- Read Error Message Code

Operationalizing evidence in the coded protocols – In this step, all the coded data were analyzed and tabulated. Final results can be seen in chapter 3 Results. This involved frequency measurement of certain codes, looking through order of code occurrence, etc.

Seeking pattern(s) in the mapped formalism – Based on the results, the pattern which emerged is Link type of error message has advantage over the Example type and Default type of error message.

Interpreting the pattern(s) - This is explained in detail in chapter 5 discussion. In that chapter with the help of different results and transcriptions, we try to explain why we observed these patterns.

Repeating the whole process, perhaps coding at a different grain size –The whole process of coding and analyzing the coded data was repeated with a different perspective – a program perspective instead of error message. This process helped us to understand whether there is any effect of the programs and their order of exposure to participants.

Chapter 4

RESULTS

4.1 Cohen's Kappa Coefficient for Inter-Rater Agreement

Once the coding was done, 3 participants data was chosen randomly which covered all the three scenarios presented in Table 1. This set was coded by two raters independently. After coding Cohen's Kappa was calculated on the coded data to measure interrater reliability. For the basic steps coding that is for marking of 'Examine/Explore/Hypothesize/Repair/Evaluate', we got a score of 0.8736 which means near perfect agreement. For the expected and unwanted steps, we got a score of 0.62 which means moderate agreement.

4.2 Time to Resolve the Error

The time to fix the error is the time from when the error message was received to when the error was fixed. Usually, even after the error was resolved a participant would perform other activities like simulating the program. Time spent on those activities is not considered. Table 6 provides the average time it took for the participants to fix an error for the different error messages types.

	Average (seconds)	STD	Variance
Default	184.34	148.77	22132.06
Link	136.67	74.047	5482.96
Example	200.25	101.62	10325.66

Table 6-Time Taken to Resolve the Error
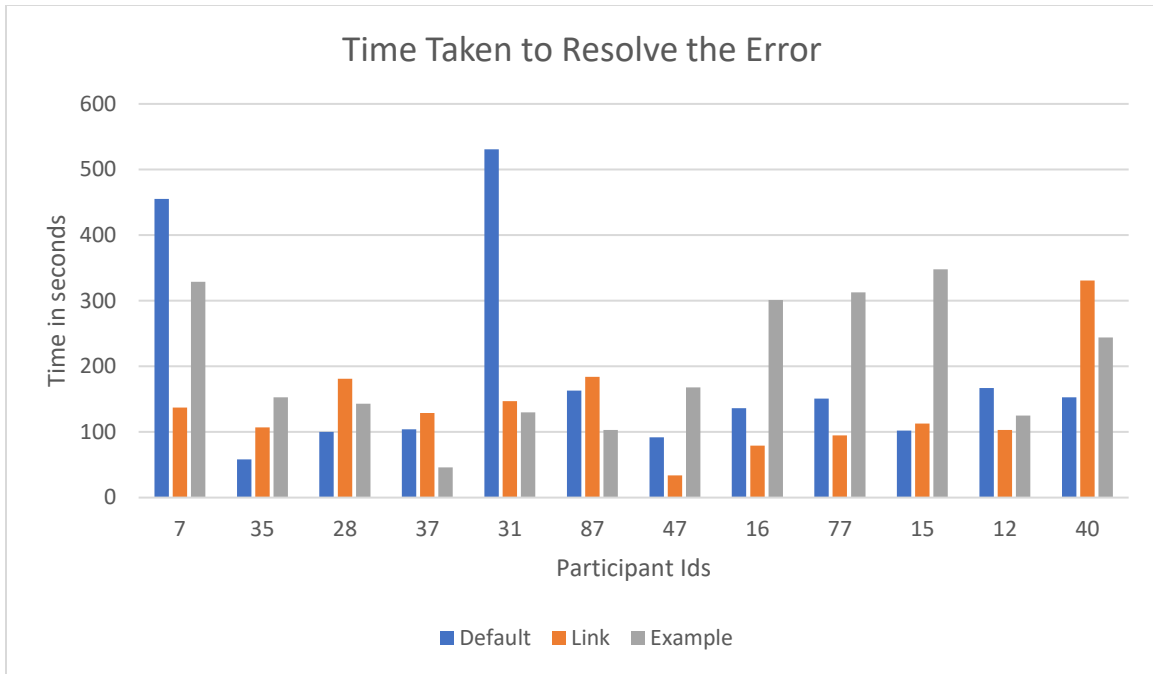
Figure 6- Time Taken to Resolve the Error

As can be seen from the table, programs which are given the *Link* type of error message took less time when compared to the other types of error message. The *Link* type took less time, the time was also more consistent when compared to other types of error message. The *Example* type took more time to fix than the default type.

4.3 Number of Steps Taken to Resolve the Error

For every program which was solved and coded, we measured the number of steps taken to fix the error. In every program, we had segmented into different steps or activities where each step is coded as one among Examine/Explore/Hypothesis/Repair/Evaluate. The table 7 gives the statistics related to steps taken to fix the error.

	Average	STD	Variance
Default	14	11.3	127.81
Link	9.8	4.17	17.42
Example	15.83	10.77	116.15

Table 7- Number of Steps Taken to Resolve the Error

28

The same pattern appears in the Table 7 as of Table 6. The *Link* type not only is taking less time to fix the error but also taking few steps when compared to other two type of error message. Again, the *Link* type is more consistent. The *Example* type is taking more steps to fix the error when compared to other two.
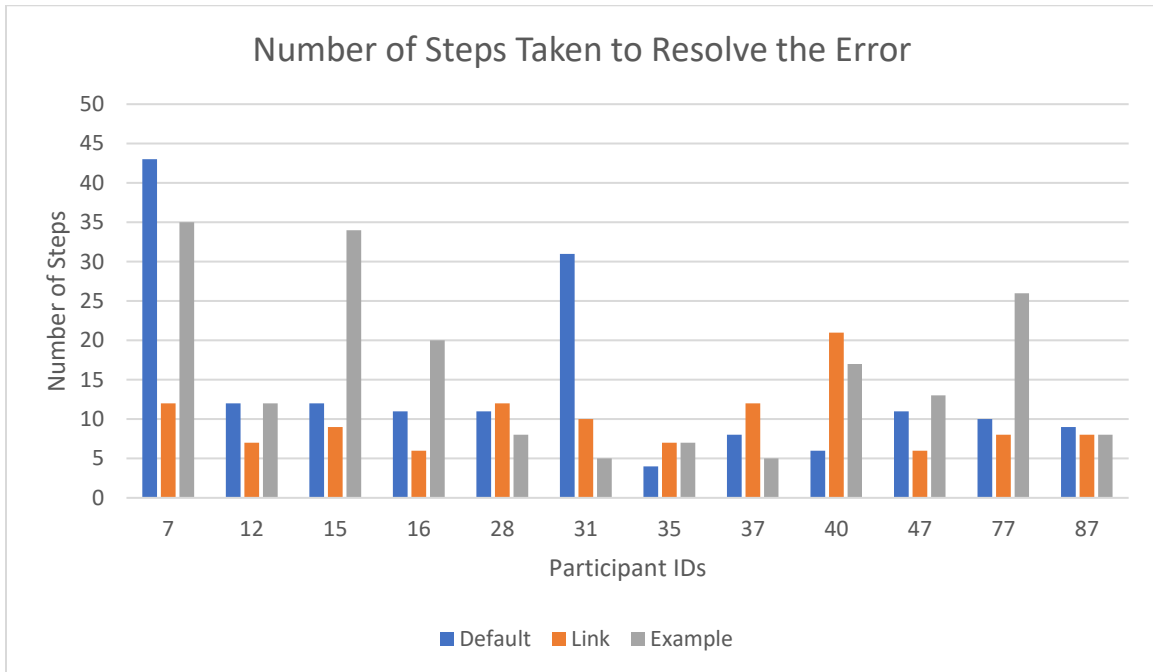
Figure 7- Number of Steps Taken to Resolve the Error

4.4 Correct and Incorrect Steps

Among Explore-Hypothesize-Repair, we can code it as correct steps or incorrect steps. Here correct steps include those which are coded as Expected or Correct Independent and incorrect steps include those which are coded as Silly Mistake, Trial & Error, Gaming, Incorrect Interpretation and Incorrect Independent. Table 8 provides how much of these steps (Explore-Hypothesize-Repair) were correct steps and how much were incorrect steps.

Based on table 8, programs given with Link type of error message has fewer wrong steps. So, based on previous data errors given in Link type of error message took less time to fix, took fewer steps to fix and among the steps taken most of them tend to be right steps.

	Correct Steps (%)	Incorrect Steps (%)
Default	54 (59.34%)	37 (40.65%)
Link	53 (89.83%)	6 (10.17%)
Example	73 (71.56%)	29 (28.43%)

Table 8- Correct and Incorrect Steps
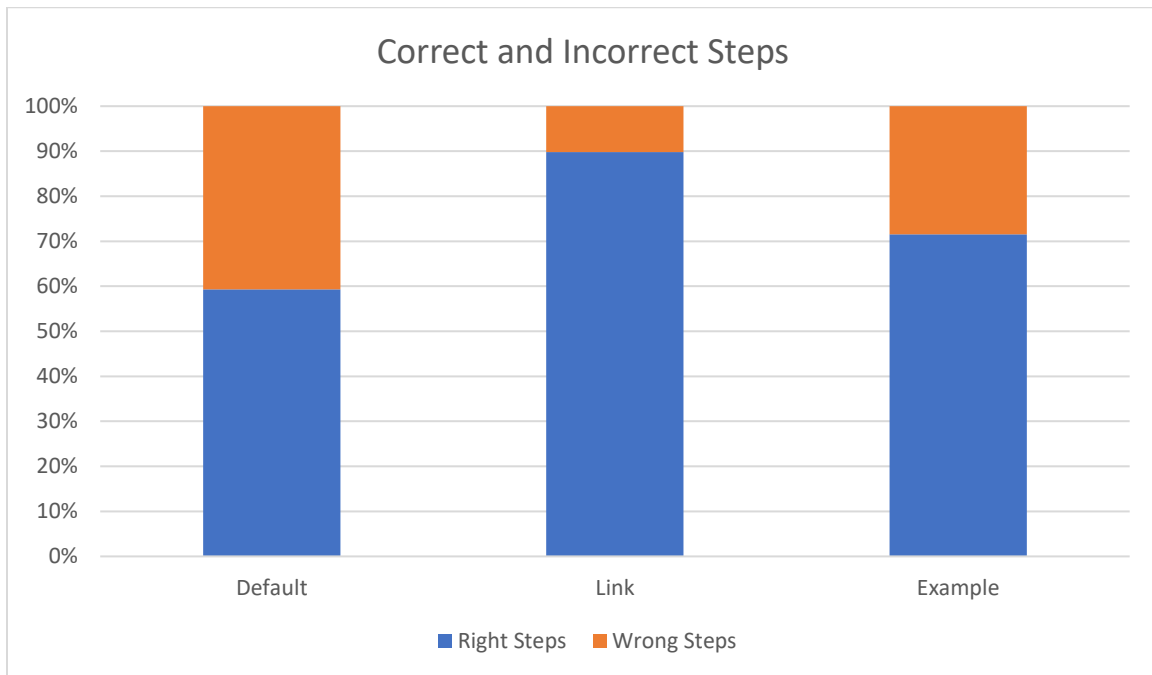
Figure 8- Correct and Incorrect Steps for different Error Message Type

4.5 Error Message Read

Table 9 gives the percentage of participants who read the error message while solving the error. Participants can read the error message partially or completely. There is also possibility of ignoring the error message.
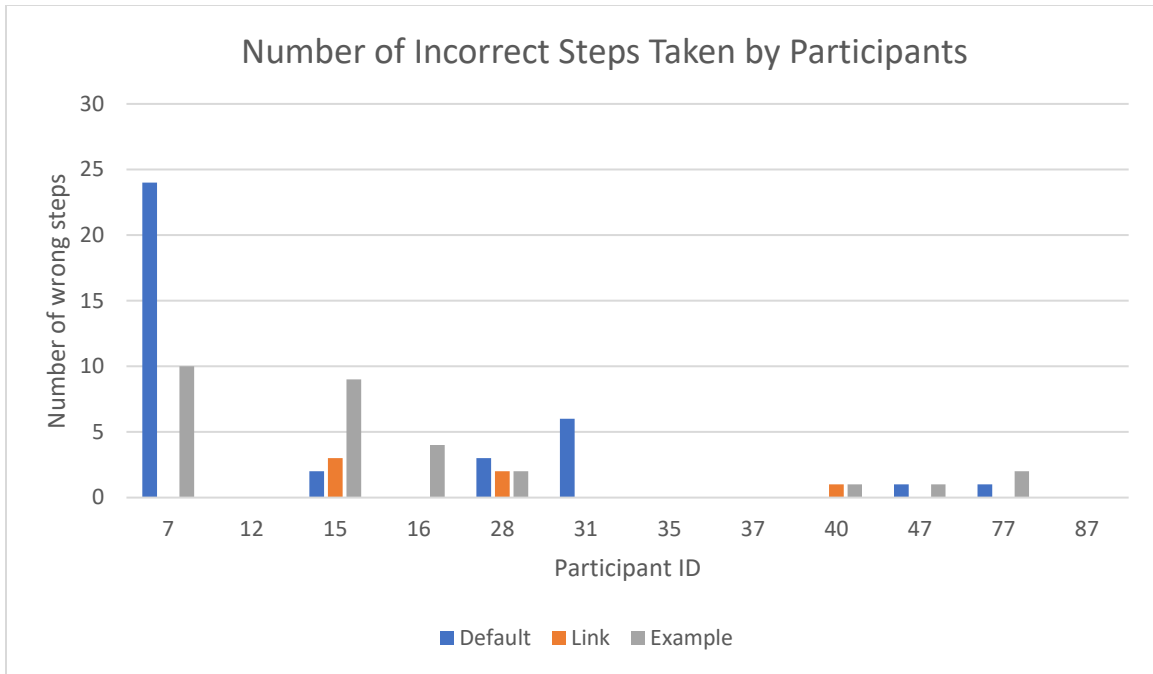
Figure 9- Number of Incorrect Steps Taken by Participants

	Complete	**Partial**	**Ignore**
Default	91.67	0	8.33
Link	66.67	33.33	0
Example	50	41.67	8.33

Table 9- Read Error Message

The *Default* type has highest percentage of complete reading of error message.

Both the *Link* type and the *Example* type have two parts in the error messages. First part

is a detailed error description and next part is either a hyperlink or a sample code. In

most of the scenarios participant only reads the description and skips the second part of

the error message. Sometimes participant may not verbalize the seen error message. In

such cases, the data is inferred based on the information given in the answers of

feedback questionnaire
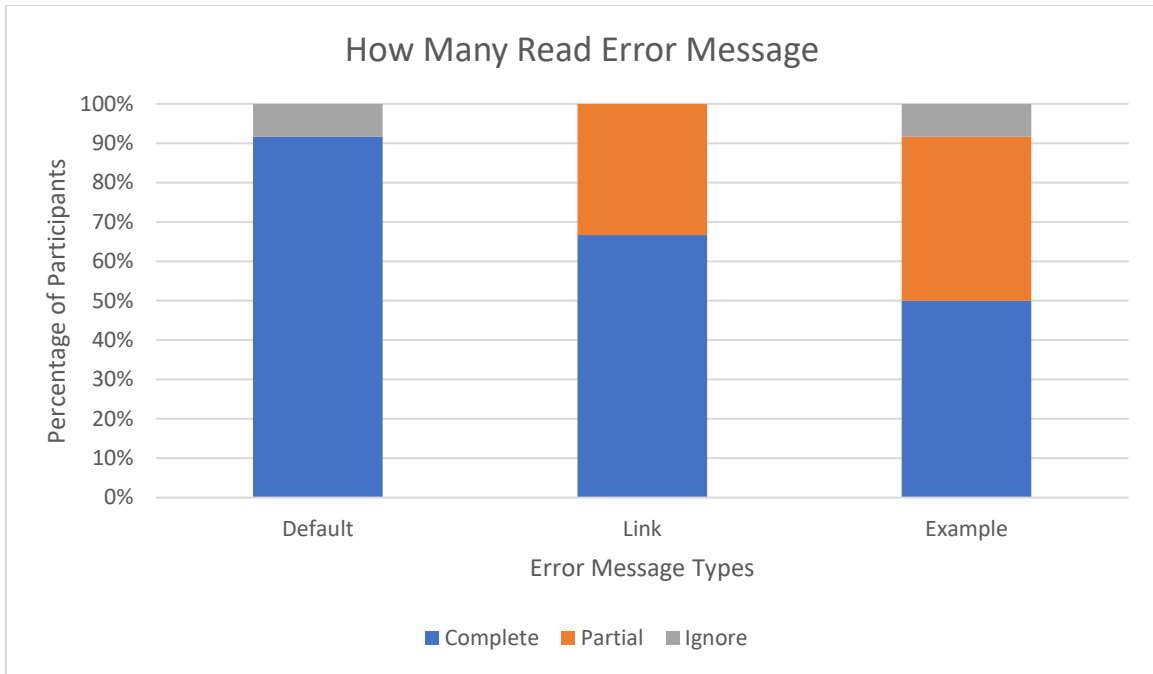
Figure 10- How Many Read Error Message

4.6 Confusion After Reading Error Message

After reading the error message, participants sometimes express confusion. There are instances where participants verbalize this confusion while solving an error or they mention it in the feedback questionnaire. Table 10 provides the percentage of participants who expressed confusion after reading an error message of type.

Default	Link	Example
16.67%	0%	50.00%

Table 10- Percentage of Participant Expressing Confusion After Reading Error Message

Participants after reading the *Link* type of error message did not express any confusion where as participants after reading the Example type of error message, maximum number of them showed confusion.
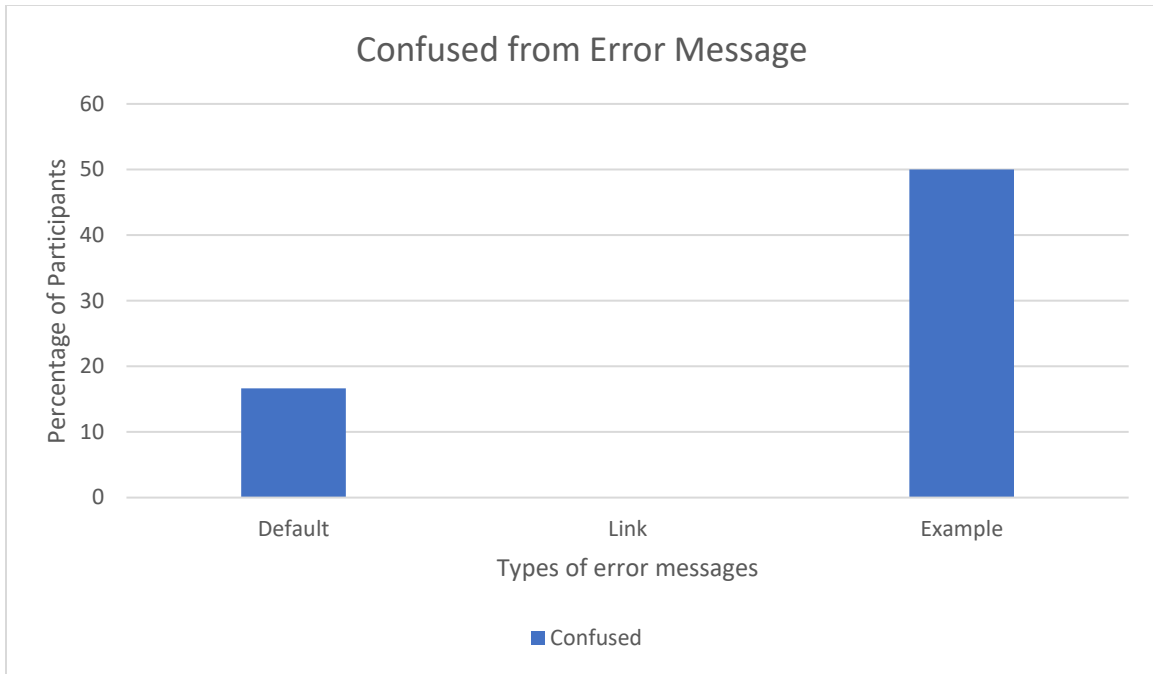
Figure 11- Percentage of Participants Confused on Reading Error Message

4.7 Percentage of Participants Using Online Manual or Quick Reference

To get extra information about the program or instruction, participants use either online manual or quick reference provided in the PLPTool itself. Table 11 gives the percentage of participants who took help from these two when given different type of error messages.

Default	Link	Example
58.33%	58.33%	50.00%

Table 11- Percentage of Participants Using Online Manual or Quick Reference

4.8 Correct Explore Steps

As seen in Table 11, irrespective of the type of error message participants have use EXPLORE step, i.e. explore online manual or PLPTool quick reference. Table 12 provides among those EXPLORE steps, how many are incorrect steps. The *Link* type has 0 incorrect steps where as other two types have some incorrect EXPLORE steps.

	Default	Link	Example
Total Explore Steps	26	14	28
Wrong Explore Steps	7	0	8

Table 12- Total EXPLORE Steps and Incorrect EXPLORE Steps
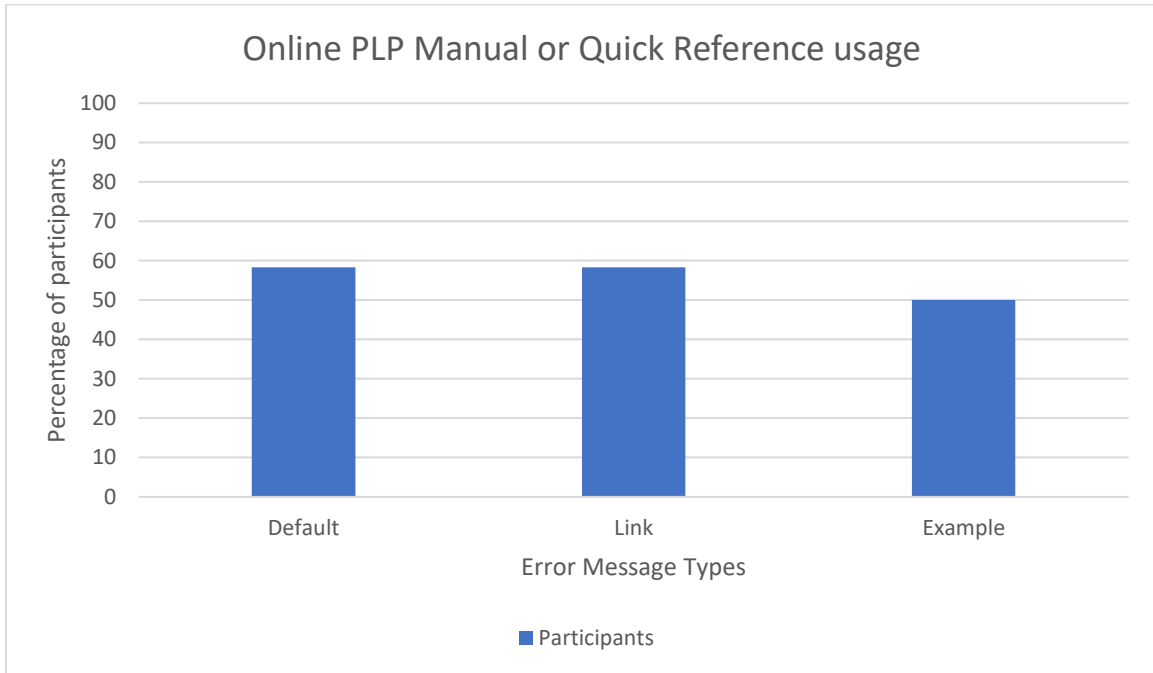
Figure 12- Percentage of Participants Using Online Manual or Quick Reference

4.9 Correct or Incorrect Understanding of Error Message

After participant reads the error message, they hypothesize the solution or express the error in their own words. If participant can understand the error message, then they hypothesis right solution or clearly express the error in their own words. Table 13 provides the percentage of correct and incorrect hypothesis immediately after reading an error message.

	Correct Hypothesis (%)	Incorrect Hypothesis (%)
Default	12 (66.67%)	6 (33.33%)
Link	20 (90.90%)	2 (9.1%)
Example	18 (78.26%)	5 (21.74%)

Table 13- Correct or Incorrect Understanding of Error Message
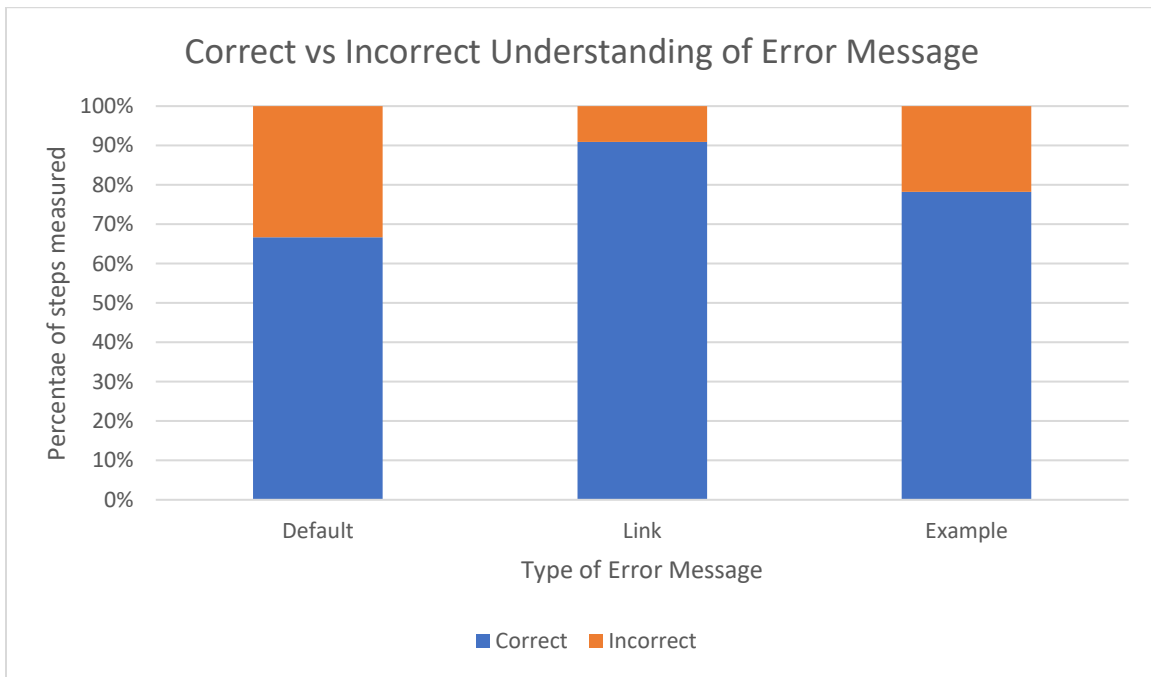
Figure 13- Correct or Incorrect Understanding of Error Message

4.10 Effect of Programs and Their Order of Presentation

All the transcribed data was recoded to understand the effect of program and
their order of exposure to student programmers. The question we are trying to answer by
following this step was whether these patterns which we have observed till now is a
resultant of the programs and their order of presentation to the participants. As the
Label program is the first program, participant should take more time and more steps to
fix it as they are still getting adjusted to the PLP, PLPTool and the experiment in general.
As the new programs are given, their performance should improve. Table 14 show that

from first program to second program there is an increase in the time taken to fix it and again from second program to third program there is a decrease in the time taken to fix the error. Similar pattern can be observed for number of steps which is shown in Table 15. Table 16 shows the percentage of correct vs incorrect steps with respect to various programs. Clearly there is some effect of the order of program as for first program (Label program) there is higher percentage of incorrect steps as shown in Table 16 even though it took less time to fix and fewer steps to fix the error. But, overall order of the programs has not contributed to the above observed pattern for error message types.
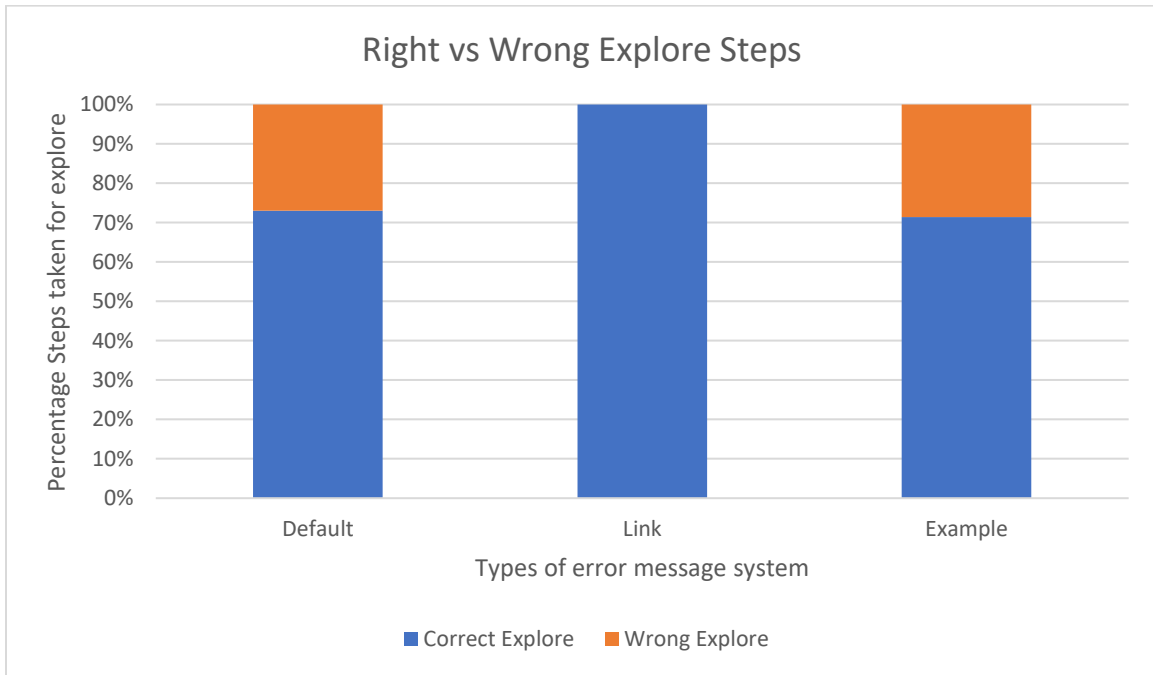
Figure 14- Total EXPLORE Steps and Incorrect EXPLORE Steps

	Average	STD	Variance
Label Program	119.38	106.71	11387.26
Instruction Program	234.46	123.09	15153.60
Register Program	157.38	70.30	4942.42

Table 14- Program Wise Time Taken to Fix the Error
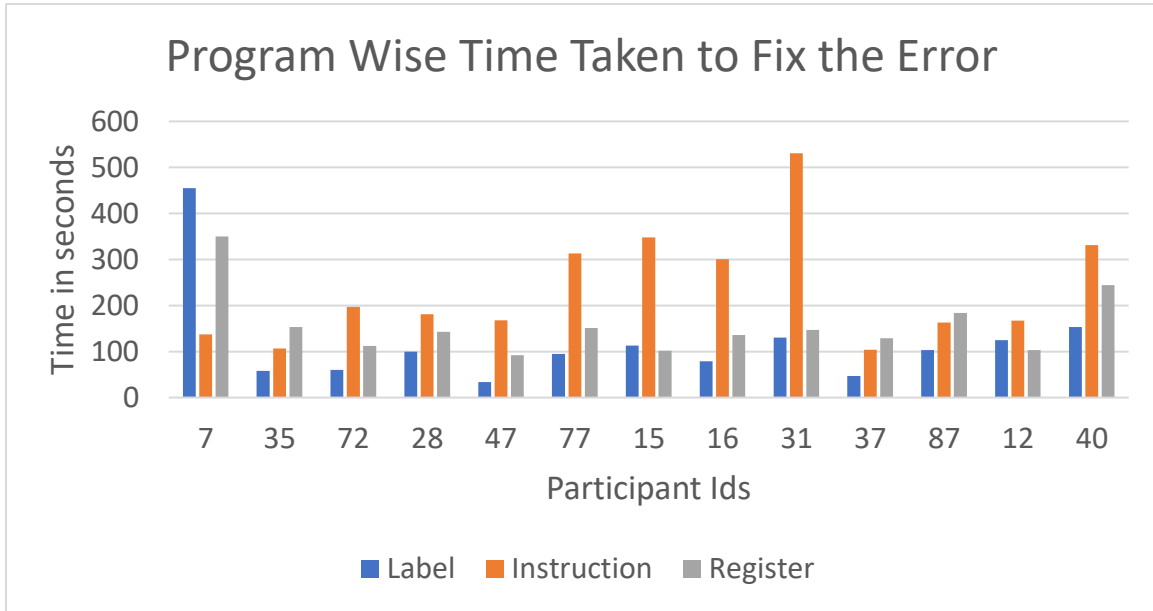
Figure 15- Program Wise Time Taken to Fix the Error

	Average	STD	Variance
Label	10.07	10.16	103.41
Instruction	17.76	8.98	80.69
Register	12.07	7.38	54.57

Table 15-Program Wise Number of Steps to Fix the Error

	Correct Steps (%)	Incorrect Steps (%)
LABEL	35 (53.03%)	31 (46.97%)
INSTRUCTION	97 (77.60%)	28 (22.40%)
REGISTER	69 (79.31%)	18 (20.68%)

Table 16- Correct and Incorrect Steps Taken Program Wise
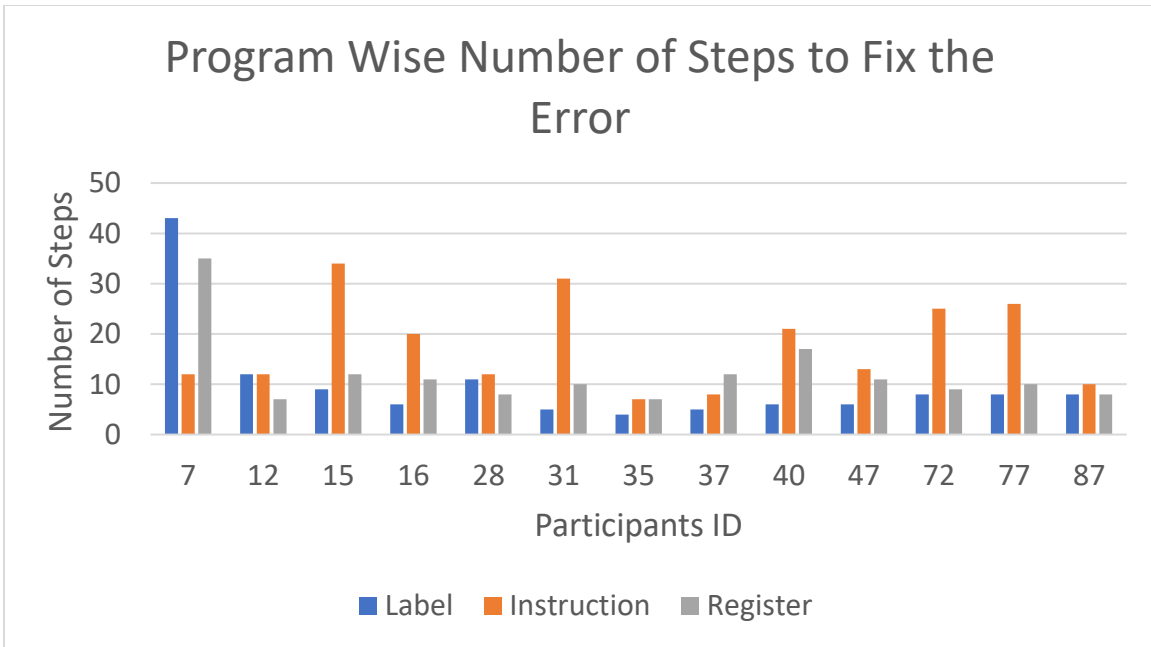
Figure 16- Program Wise Number of Steps to Fix the Error
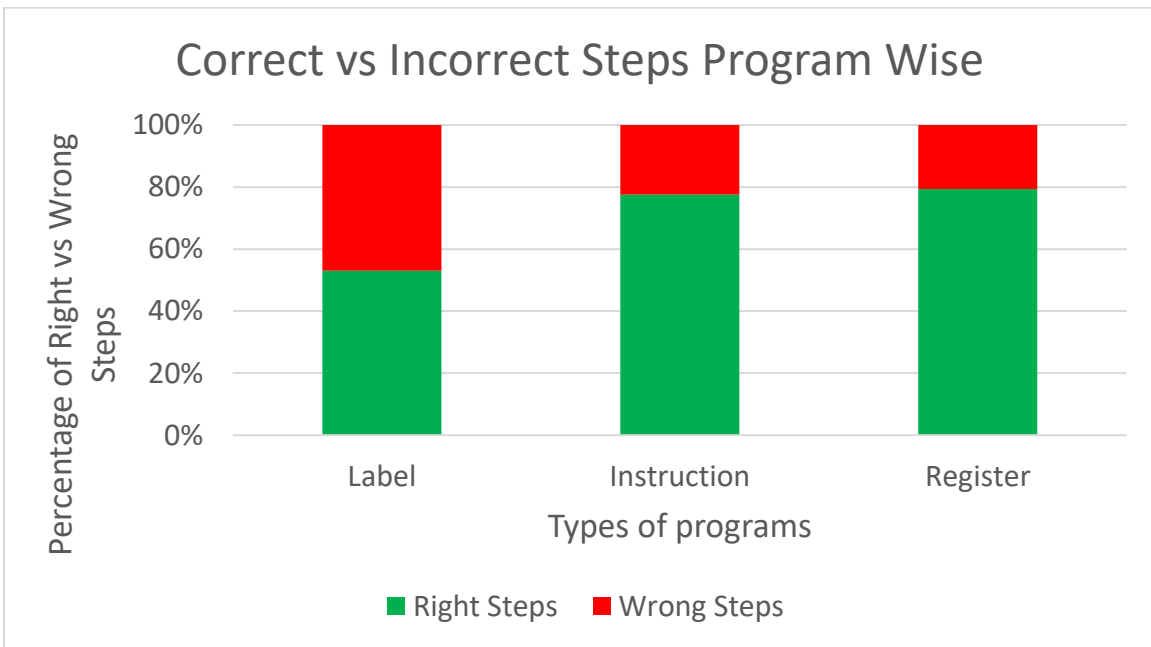
Figure 17- Correct and Incorrect Steps Taken Program Wise

Chapter 5

DISCUSSION

5.1 Descriptive Analysis

Based on the data we have collected and analyzed, it is very clear that the *Link* type of error message system helped the most for the participants. Now let us try to understand how did the *Link* type of error message helped participants to understand the error and how did it help them to fix it.

What aspects of an error message help the programmer to understand the error?

First thing which helps a participant in trying to know about the error is description of the error message. The *Link* type when compared to the *Default* type had a detailed description. The error description of the *Link* type was generated based on the program's perspective rather than assembler's perspective. For both the *Label* program and the *Instruction* program, the *Default* type showed similar description of error message. Even though both are completely different errors, for an assembler theses errors are caught at the same stage. Hence the *Default* type gave similar description for both errors. For the *Label* program, the *Default* type error message had description "unable to process token addition" and for the *Instruction* program, the *Default* type error message had description "unable to process token subiu". These error message descriptions have words like 'token' which are related to assembler functioning and it has less information to the programmer. If a student does not know the working of assembler and is a novice, then "unable to process token" would be alien to his current knowledge model. So, error message is not helping the programmer to understand what is the error.

When programmer don't understand the error, they tend to use either TRAIL &
ERROR or GAMING way of fixing the error. This can be seen in case of participant
##7## for whom the *Default* type error message was given for the *Label* program.
Participant after reading the error message starts changing code at places where word
'addition' appears even though that line has nothing to do with error. Participant has
committed 5 TRAIL & ERROR code changes, 2 INCORRECT INTERPRETATION code
changes and 5 INCORRECT INDEPENDENT code changes. Even though participant
corrected the error but does not show any signs that participant understood the meaning
of the error. For the feedback question 'could you explain in your words what was the
error in the program' the participant answers as "*There was a missing ":" at the end of
the function definition. This made the function incomplete.*" This clearly shows
participant did not understand what is the error.

Good Amount of Error Information in the Error Message

Second problem with the *Default* type of error message is it has too little
information about the error. As there is too little information, programmers had to take
extra steps to get more information and understand better about the error. To get these
extra information, they usually search in internet, use online PLP manual or PLPTool
quick reference. For participant ##12##, Instruction program was provided with Default
error message type. The error message read "unable to process token subiu". After
reading the error message, participant opens the quick reference and tries to search for
subiu. Participant is unable to find subiu, then makes a connection that there is no such
instruction. Here is the transcript of participant ##12##, "*it says unable to process
token subio subiu okay ahhn so I will just go to the help quick reference check for
subiu... so I can see that there is no subiu operation given in the quick reference card
this means that there is no such token...*". This behavior can be seen in other participants

like participant ##31## *"error is unable to process token subiu then it will jumping on exit… [long pause] this is a right command right subiu?... So... I am just checking the quick reference and making sure that subiu is the right command I found addiu but hmmm so addiu is for sign extended addition but there is no I am not able to find any command for subtraction..."*

Due to insufficient error information, programmer might take some time or some more steps before finally understanding what is the error. Participant ##28## was provided with Default type of error message for Label program. ##28## after reading the error message for the first time interprets it incorrectly. After reading the error message which was generated due to wrong code correction, comes to the right understanding of the error. Transcript of participant ##28## is *"after assembled I am getting error at main asm 20 unable to process token addition line 20 that would be ahmm it says addition on that line and it appears it could be either a comment I am not understanding exactly what the user was attempting to do since there is an addition below it were in addu operation below it I am going to comment out addition word and see if it should be a jump nope nope since it is a last one see if that works save reassemble okay so next error says asm 13 invalid branch target addition so that based on line of 13 addition is no supposed to be comment it supposed to be a label..."*

In case of the *Link* type and the *Example* type of error message, description of the error is descriptive and formed with respect to the program which is being assembled. For the *Label* program, description of error by the *Link* and the *Example* type is "The error caused due to missing colon : after label name addition. May need a : at the end to declare it as label. Error occurred around word addition." Similarly, for the *Instruction* program, description of error by the *Link* and the *Example* type is "The error is caused due to wrong instruction name. PLP does not have an instruction as mentioned in the

code. subiu – instruction is not defined in PLP. Unable to process the instruction. Error occurred around word subiu". In both the error descriptions, there are no words which are related to assembler or its inner working.

As sufficient information is provided with the error description, participants would understand the error without need to search the extra information. As the message is clear, they would interpret the error message in the right way. Let us consider the *Label* Programs error message provided with the *Link* type or the *Example* type of error message. Participant ##16## reads the error message and then hypothesizes what is the error followed by correct code change to fix the error. Transcript - "*Okay I see that there is an error on line 16 so that's what I am looking at line 20 I mean with the addition… okay so in addition I see that there is it is prompting me that error is in line 20 there is a missing colon So I am going to try adding that making it a label and see if that fixes it*". Clearly participant knew that problem is caused due to incorrect label declaration. Let us see participant ##37## transcription – "*the error is caused due to missing semi colon after label name addition okay [inaudible words] oh so I want the semi colon after the label name addition so that's it this is a method and I want to go ahead and add the colon for that so that's a label and label name addition oh it even told me the method interesting okay at the end to declare its label. Error occurred around word addition*". Here also participant ##37## clearly made the connection that error is related to incorrect label declaration. Similarly, participant ##47## understands that error is caused due to incorrect label declaration – "*okay asm 20 the error is due to missing colon lets see main 20 exit j oh j exit now I can see what it is doing ahhh chuchu may need a colon at the end of declare as label Oh! Yeah that's it it just a colon at the end it will make it a label so… compile it*". There is a possibility of gaming with this error message description as solution for the fix is given directly in the error description. So

42

even without understanding the error, participant just try to do what the error description says and might get it correct. This behavior was observed in one participant out of twelve participants.

Let's consider the error message given for the *Instruction* Program using the *Link* type or the *Example* type. The error description clearly mentioned that subiu was not an instruction and hence participants next step was not to ascertain whether there is an instruction called subiu but rather how to fix this. Let's see the transcript of participant ##7##, the same participant who failed to understand the error in the *Label* program. In this case, error was displayed using the *Link* type of error message. Here is the transcript - "*error caused by wrong instruction name PLP does not have an instruction as mentioned in the code. subiu instruction is not defined in PLP unable to process the instruction error occurred around word subiu alright [reads complete error message]... so what about do this I deleted I from subiu...[does the code changes and compiles] to see if compiles again alright now register used is not recognized by the PLP subu is expecting a register but got something else 10 Okay you can't subtract a number you can't use a number when you using subu so I go and declare a register and store a number in it*". After reading error message participant is thinking of fixing of error message rather than checking whether subiu is an instruction or not. Similar case happens for participant ##47##. ##47## was provided with the *Example* type of error message for Instruction program. In this case participant reads the error message partially, just the description and ignores the sample code part. Transcript is "*okay error is caused by wrong instruction name PLP does not have an instruction as mentioned line 16 number 16 it line 14... Oh subiu it should be subu there is no instruction*". Let's see the transcript of participant ##28## who was given the *Link* type of error message in the *Instruction* program. Transcript is "*rest of the error says this error is caused due to*

wrong instruction name. PLP does not have an instruction as mentioned in the code subiu instruction is not defined in the PLP Unable to process the instruction error occurred around subiu please refer following link for more information okay iu does not sounds right it should probably ui I believe... ahmmm.. unsigned immediate yeah subu unsigned immediate lets give that a shot". Even though ##28## interpretation for fix is wrong, but after reading the error message next step was not to ascertain whether there is an instruction called subiu is there or not rather the thought process was how to correct the wrong instruction of subiu.

Once participant understands the error, they either rephrase the error as understood by them or verbalize their way of solving the error. This is marked as HYPOTHESIS step in our coding. If participant immediately after reading an error message HYPOTHESIS, then that step gives some insight into participants understanding of error based on the given error message. Table 13 gives the percentage of those HYPOTHESIS steps which are correct and incorrect among different types of error message. The *Link* type has higher ratio of correct HYPOTHESIS steps when compared to others. The *Default* type has the least favorable ratio when compared to other two types.

So, with a detailed error message description which describes what is wrong based on programs perspective rather than at what stage of assembling did program fail we are reducing the number of steps taken to understand the error. As this kind of error message help them to understand the error better, programmers tend to avoid the TRAIL & ERROR or GAMING way of fixing the error. Thus, again leading to overall less time and overall few steps for error correction.

*What aspects of an error message help the programmer to fix the error?*The *Default* type of error message has just a small description usually explaining at which stage of assembling did the program fail. Some error messages mentioned the exact word at which point error was generated. Apart from that there was not much help was provided to correct the error. For the Register program, the *Default* type of error message provided the message "Invalid Register(s)", but did not mention which register was invalid. In the instruction which was having the error there were 3 registers. Users had to verify each of the registers to check which register was invalid and which others are valid. Participant ##15## was provided with the *Default* type of error message for the *Register* program. Here is the transcript – "*Okay so I can see two ahh one error in it at main.asm 28 invalid register so tells me that there is a invalid register used in this program at line number 28 so its like ahh $v2, $a1, a0 jr ra so okay I have to figure out which register it is that is invalid*". Clearly more steps and time is spent.

Without better understanding of the error it is difficult to come up with a proper fix for the error. To solve the poor understanding problem in the *Default* type, both the *Link* type and the *Example* type of error message system had detailed description. Even though both had similar error description message, the *Link* type has better performance when compared to the *Example* type. In fact, in some cases the *Example* type performance is worse than the *Default* type.

Why does the Link type perform better than the Example type?

The *Link* type provided a hyperlink to a section of online PLP manual, where participant can get more information about the instructions which had errors in the program. By reading more about the instruction and its syntax, programmer would be able to figure out exact reason of error and which in turn help them to fixing that part

(Lazonder, A. W., & van der Meij, H., 1995). The *Example* type provided a relevant example code with a format of "BEFORE CORRECTION sample instruction AFTER CORRECTION sample instruction", where a similar code error was provided as example and how the fixed code looked like (Hartmann, B., MacDougall, D., Brandt, J., & Klemmer, S. R., 2010, April). By looking at this example code, programmer could relate to the error they are facing. By relating it they should now be able to better understand it and thus fix the error like how the example code fix is provided.

Let us consider reasons why the *Example* type error message failed to give similar results as the *Link* type. As can be seen in Table – 10 about 50% of the participants expressed or showed confusion after reading error message of the *Example* type. Similarly, 16.67% of participants expressed or showed confusion after reading error message of the *Default* type. For the *Default* type as there was very little information was provided and message told at what stage of assembling error occurred instead of explaining it in program view, there could arise confusion. What part of Example type was confusing? It is not the error description. If error description was confusing, then even Link type should have created the confusion among the participants as both the *Link* type and the *Example* type share same error description. The main source of confusion is the example code given in the *Example* type. Participants after seeing the example code, instead of relating the code to the current error, participants thought that example code is part of the program, not an example and that is causing the error. Participant ##72## expresses this - "*the error says the issue is on line number 28 the register used is not recognized by the PLP mullo is expecting a register but got something else $v2 error occurred around word $v2 if I click error location which mentioned in the console it will take me to the line where error is there and saying $v2 is not supported $v2 is the it also tells that before correction addition operation is like*

this and after correction addition operation is like this but we are not using addition operation here". Here error was caused due to wrong usage of register. The line had mullo $v2, $a1, $a0 and sample code was provided as "BEFORE CORRECTION addu $t1, $t2, $a4 AFTER CORRECTION addu $t1, $t2, $a3". The reason this example was given is like how $a4 was replaced with $a3 in addu instruction, participant should replace $v2 with some other register in mullo instruction. But participant thought that sample code was part of program. Participant ##28## expressed similar confusion. ##28## in feedback explains the confusing part of Example type error message – "*the first part of the error message was clear and concise leading me to the correct line of code and even the correct register, however, the sample fix provided was confusing because it did not match the exact register, or line, and so I have a moment of confusion*".

Some participants after experiencing the confusion, will ignore the sample code, and try to solve it just using the description, but many of the participants due to this confusion starts committing other errors at wrong locations. Participant ##16## was given the *Example* type of error message for the *Instruction* program. Transcript of participant ##16## - "*so I tried changing it to subu to see if it fixes it... okay that fixed that error but there is another error with the add instruction... this one based on the error message it is little harder to figure out what it is I think... I think it is because with the register I think... so that ahmmm... this one it says before and after correction ahmmm.... It just kind of confusing error message because I haven't changed there so I will try addu and see if it does anything.*" ##16##, because of this confusion, started misinterpreting the error message and did the code changes at wrong locations thus resulting in more unwanted errors and steps. ##15## did similar mistake of misinterpreting the error message of the *Example* type for the *Instruction* program

47

which resulted in 8 unwanted steps which included TRAIL & ERROR, GAMING, INCORRECT INTERPRETATION. Even participant ##77## got confused with the *Example* type error message for the *Instruction* program resulting in 2 extra errors.

Even if participant understands that given sample code is an example, it takes some steps or time to understand what is that example code is conveying and how is that related to current error. Participant ##40## displays this behavior in the Register program where Example type of error message was displayed – "*whats this before correction and after correction in the console… Okay before correction addu $t1 $t2 $a4 after correction addu $t1 $t2 $a3 lets see… [looks in online manual for notes on register $a0-$a3] so for a its a0 to a3 only Okay ahhn if I change it to v0*". Particpant ##40## uses online manual to understand about sample code and then based on that deduces that v0 should be used for fix. Thus, increasing the number of steps and time for fix. Out of 12 participants, 4 of them read the *Example* type error message partially. They read only the description ignoring the example code part. Among them total of only one incorrect step is observed.

How do the Link type error message provide better performance?

Table – 11 gives the percentage of participants who used either online PLP manual or quick reference while solving the error in the programs. Even though we are only providing hyperlink in the *Link* type of error message, online PLP manual and quick reference is accessed almost equally for other types of error message system. Then, we need to ask the question what is the added advantage the *Link* type of error message system is providing? Table 12 gives answer to this question. In case of the *Default* type and the *Example* type of error message, participants should search the online manual or the quick reference in-order to come to the relevant information. Sometimes due to

confusion after reading the error message or due to little information provided by the error message, participants are not sure what to search for in the online manual or quick reference. So, they look for few other information before actually consider relevant information. Thus, increasing the number of steps to solve the error as well as increasing the time to fix the error. In case of the *Link* type of error message, participant is clearly given the location where they need to look for the right information. They need not have to spend time in searching for that. These links also act as a confidence building to the hypothesis of possible cause of error. Online PLP manual has clear cut explanation of each instruction, its syntax as well as some code examples of how they should be used. In case of the *Instruction* program where error is due to subiu an invalid instruction, the *Link* type provides hyperlink to arithmetic instructions section of the PLP online manual. By going through that, participant not only understands that there is no instruction called subiu but also learns about the syntax of the alternate instruction subu which needs to be used. The error instruction was subiu $s1, $t3, 10. Now just by replacing subiu with subu is not enough, we need to replace 10 with another register as subu expects all inputs in registers. This information is clearly provided in the manual. In case of the *Register* program where error is due to $v2 an invalid register, the *Link* type provides hyperlink to register usage convention section of PLP online manual. Here also participant not only confirms that there is no $v2 listed in the PLP registers but also finds the complete list of registers available in PLP. So, participant can then decide which one to use based on the convention.

Thus, the *Link* type provides a detailed program-centric error description to better understand the error. By doing this it is avoiding the trial & error and gaming way of fixing the error. Then it provides hyperlinks to online manual which not only further improves in understanding of error but also gives proper syntax and other information

49

which help participants to fix the error by reducing the chance of incorrect interpretation of the error message.

5.2 Reflection on Experiment

All the participants could fix errors in all the programs given to them. Though during the process some of them committed other errors due to wrong code changes. Once recording has been stopped that is once experiment is over, participants tend to involve in informal talks and discuss about given programs, errors they encountered and error messaging types. They give valuable feedback during that time. A trial run of the experiment was conducted before recruitment advertisement for participation was made public. This helped to identify few gaps in the experiment setup like environment setup for the experiment which helped to conduct experiment with other participants smoothly. In the demographic survey, most of the participants choose PLP proficiency as intermediate. It would be better to have a 4-scale or even a number ranking scale as options for the questions in demographic survey. Many of the participants would become silent and get completely involved in solving the error mentally. In that scenario, they need to be prompted for "Keep talking", so that they get back to the think out aloud setup. For the feedback questionnaire, as the questions were phrased as "How did the error message helped in understanding/fixing the error?", participants tend to give only positives about the error message. It is better to mention to participants that both positives and negatives about the error message can be provided as feedback. Most of the participants finished the feedback within 2 minutes by writing just one or two lines. Participants had to be reminded that until minimum 5 minutes is over, they cannot go on to next segment. Participants were reminded of using that extra time to rephrase and add new information to the feedback.

5.3 Subjectivity

As there was no study done previously to understand the types of error which is affecting most to the students of PLP language, types of error for this study were chosen based on the consultation with the previous TA's of SER 250 course and the instructors of that course. Similarly, we need to understand how the programs were challenging to all types of students – expert and novice.

5.4 Future Work

There is still lot of data from this experiment which needs to be further analyzed. One interesting data to look for is the confidence with which a participant makes the changes to the code. We need to understand what factors help to increase the confidence of a programmer. To understand this, we need to look the data as set of impasses and their resolutions. We should also further analyze the wrong steps taken by the participants. Analysis on what type of wrong steps are taken more common and once wrong step is taken, how the programmers correct it. This could help in understanding features of the programming language and teaching techniques that negatively impact student programmers mental model. There is also needed to analyze the flow of steps taken by the participants – is there any particular pattern in debugging. We must also understand what part of the program is more examined and is there a pattern in that also – description of program, program's code, error message itself. We need to understand better about the relationship between wrong steps taken, error message and PLPTool itself. This would help us to understand the factors of a programming language, IDE and error message which would make programmer to take wrong steps and help us to remove or reduce those factors.

We need to see the effects of error message types on the novice programmers. In the current experiment, participants were selected in general without any PLP

proficiency level precondition. It is difficult to analyze how these error messages would affect the novice programmer's performance. So, in future experiment we need to have a participant pool with equal representation of novice and expert programmers. This will help to clearly understand what factors of error messages help novices and expert and find out whether it differs. As every participant in this experiment could solve the programs, we need to come up with better programs which can test more factors of the error message. One of the way is single type of error is provided in two different locations in a program with different instructions. Thus, we can see how a programmer applies learned knowledge of previous error correction. By this we can clearly say whether a programmer learned from his/her mistakes and is able to reapply those knowledge for future errors.

As for the tool and error message system itself, the current experiment is just a preliminary step towards the development of an intelligent IDE. These IDE's should act as coding assistant which improve the performance of the programmers by reducing the unnecessary steps taken by programmers. Already there are couple of researches which are moving in that direction (Layman, L. M., Williams, L. A., & St Amant, R. (2008, May); Layman, L., Williams, L., & Amant, R. S. (2007, September)). They should also teach programmers more about the programming language itself through the error messages.

Chapter 5

CONCLUSION

This exploratory study used think aloud programming exercise to understand the effects of an error message on the overall debugging process of student programmers. As part of this study, three different error message types were developed. The effect of each of these error message types was analyzed from the perspective of understanding of an error and resolving the error. In the study, it was found that an error message type which describes the error in a program-centric way rather than an assembler-centric way helped in better understanding of the error. An error message type which has a hyperlink to a relevant section in an online manual, helped in fixing the error by reducing the incorrect interpretation of the error and increasing the understanding of the error. The *Link* type of error message helped to fix the error in less time and with fewer steps. It also helped the student programmers to search for information in the online manual and quick reference in a more focused manner. On the other hand, in the study it showed that giving a relevant example code as part of the error message may lead to confusion among the student programmers. The error message which provides little description, or an assembler-centric error description also creates confusion. The confusion and lack of information in the error messages led to more steps and more time to resolve the error. This lack of clarity in the error message led to less understanding of error which in turn led to more incorrect steps in resolving the error. In this study, the *Link* type error message which has a program-centric enhanced error description and a hyperlink to relevant online manual content, appears to be an ideal error message display system.

53

REFERENCES

Chabert, J. M., & Higginbotham, T. F. (1976, April).

An investigation of novice programmer errors in IBM 370 (OS) assembly language. In *Proceedings of the 14th annual Southeast regional conference* (pp. 319-323). ACM.

Nienaltowski, M. H., Pedroni, M., & Meyer, B. (2008, March).

Compiler error messages: What can help novices? In *ACM SIGCSE Bulletin* (Vol. 40, No. 1, pp. 168-172). ACM.

Marceau, G., Fisler, K., & Krishnamurthi, S. (2011, March).

Measuring the effectiveness of error messages designed for novice programmers. In *Proceedings of the 42nd ACM technical symposium on Computer science education* (pp. 499-504). ACM.

McLaren, B. M., DeLeeuw, K. E., & Mayer, R. E. (2011).

A politeness effect in learning with web-based intelligent tutors. *International Journal of Human-Computer Studies*, *69*(1), 70-79.

Vessey, I. (1985).

Expertise in debugging computer programs: A process analysis. *International Journal of Man-Machine Studies*, *23*(5), 459-494.

Chi, M. T. (1997).

Quantifying qualitative analyses of verbal data: A practical guide. *The journal of the learning sciences*, *6*(3), 271-315.

Sohoni, S. (2014, June).

Making the hardware-software connection with PLP. In *Proceedings of the 2014 conference on Innovation & technology in computer science education* (pp. 324-324). ACM.

Lazonder, A. W., & van der Meij, H. (1995).

Error-information in tutorial documentation: supporting users' errors to facilitate initial skill learning. *International Journal of Human-Computer Studies*, *42*(2), 185-206.

Hartmann, B., MacDougall, D., Brandt, J., & Klemmer, S. R. (2010, April).

What would other programmers do: suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 1019-1028). ACM.

Traver, V. J. (2010).

On compiler error messages: what they say and what they mean. *Advances in Human-Computer Interaction*, *2010*.

Hughes, J., & Parkes, S. (2003).

Trends in the use of verbal protocol analysis in software engineering research. *Behaviour & Information Technology*, *22*(2), 127-140.

Van Someren, M. W., Barnard, Y. F., & Sandberg, J. A. C. (1994).

The think aloud method: a practical guide to modelling cognitive processes Academic Press. *London, UK*.

Ericsson, K. A., & Simon, H. A. (1993).

Protocol analysis. Cambridge, MA: MIT press.

Rodrigo, M. M. T., & Baker, R. S. (2009, August).

Coarse-grained detection of student frustration in an introductory programming course. In *Proceedings of the fifth international workshop on Computing education research workshop*(pp. 75-80). ACM.

Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L., & Zander, C. (2008).

Debugging: finding, fixing and flailing, a multi-institutional study of novice debuggers. *Computer Science Education*, *18*(2), 93-116.

Letovsky, S. (1987).

Cognitive processes in program comprehension. Journal of Systems and software, 7(4), 325-339.

Jeffries, R. (1982, March).

A comparison of the debugging behavior of expert and novice programmers. In Proceedings of AERA annual meeting.

Layman, L. M., Williams, L. A., & St Amant, R. (2008, May).

MimEc: intelligent user notification of faults in the eclipse IDE. In Proceedings of the 2008 international workshop on Cooperative and human aspects of software engineering (pp. 73-76). ACM.

Layman, L., Williams, L., & Amant, R. S. (2007, September).

Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools. In Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on (pp. 176-185). IEEE.

APPENDIX A

PROGRAMS, ERRORS, AND FIXES

As part of this study, three programs were used in the experiments. In this appendix, all the programs their errors and how those errors can be fixed are provided.

1. Label Program

    This was the first program given in the experiment to a participant. Figure 18 shows the program. This program has an error at line number 20. Here word 'addition' is a label. It is missing a ':' to declare it as label.

```
main.asm <main program>                                          Editor Mode
 1  |
 2  #Problem - Given two numbers(45, 60),based on the value in register
    $t1 perform either addition or subtraction operation.
 3  #              Final result should be stored in register $s1.
 4  #              0 - Addition
 5  #              1 - Subtraction
 6
 7  .org 0x10000000
 8
 9  li $t1, 0
10  li $t2, 45
11  li $t3, 60
12
13  beq $t1, $0, addition
14  nop
15
16  subu $s1, $t2, $t3
17  j exit
18  nop
19
20  addition
21      addu $s1, $t2, $t3
22
23  exit:
24      j exit
25      nop
```

Figure 18- the Label Program

Figure 19 gives the error message as displayed using the *Default* type.
Figure 20 gives the error message as displayed using the *Link* type and Figure 21
gives the error message as displayed using the *Example* type. The correct way to
fix this error is to place a ':' at the end of line 20 to make word addition as label.
So, line 20 will have 'addition:' after error correction.

```
[ERROR] #16 Asm: main.asm:20: Unable to process token addition
[ERROR] #24 Asm: preprocess: 1 error(s).
```

Figure 19- Default Type Error Message for the Label Program

```
[ERROR] #16 Asm: main.asm:20 This error is caused due to missing colon ":" after label name.
addition - may need a ":" at the end to declare it as label.Error occurred around word - "addition"
Please refer following link for more information http://progressive-learning-platform.github.
io/instructions.html#labels
[ERROR] #24 Asm: preprocess: 1 error(s).
```

Figure 20- Link Type Error Message for the Label Program

```
[ERROR] #16 Asm: main.asm:20 This error is caused due to missing colon ":" after label name.
addition - may need a ":" at the end to declare it as label.Error occurred around word - "addition"

BEFORE CORRECTION
End_Loop
j loop_begin
AFTER CORRECTION
End_Loop:
j loop_begin
[ERROR] #24 Asm: preprocess: 1 error(s).
```

Figure 21- Example Type Error Message for the Label Program

2. Instruction Program:

This was the second program given to participants during the experiment. Figure 22
gives the *Instruction* program. This program has an error at line 14. Here program's
intention is to subtract value 10 from the register $t3 and store back the result into
register $s1. For unsigned operations, we do not have a subtract instruction in PLP.

So 'subiu' is not an instruction in PLP. Figure 23 gives the error message as displayed by the *Default* type. Figure 24 gives the error message as displayed by the *Link* type and Figure 25 gives the error message as displayed by the *Example* type.

```
main.asm <main program>                                          Editor Mode

 1
 2     # Problem - Given a number (60), based on the value in register
       $tl, perform either increment by 10 or decrement by 10.
 3     #              Final result is stored in register $sl.
 4     #                 0 - Increment Operation
 5     #                 1 - Decrement Operation
 6
 7     .org 0x10000000
 8
 9     li $tl, 1
10     li $t3, 60
11
12     beq $tl, $0, increment
13     nop
14     subiu $sl, $t3, 10
15     j exit
16     nop
17
18     increment:
19             addiu $sl, $t3, 10
20
21     exit:
22             j exit
23             nop
```

Figure 22- the Instruction Program

There are two ways of fixing this error. First way is using subu instruction of PLP. As subu cannot have an immediate value, we need to first load the value 10 into a register say $t2, using li instruction. Use the newly loaded register in subu instruction to achieve the decrement operation. So, line 14 can be replaced with these two lines 'li $t2, 10' and 'subu $s1, $t3, $t2'. Second way is using addiu instruction of PLP. Replace line 14 with 'addiu $s1, $t3, -10'. As you can see in both the ways there is steps to fix the error.

59

```
[ERROR] #16 Asm: main.asm:14: Unable to process token subiu
[ERROR] #24 Asm: preprocess: 1 error(s).
```

Figure 23- Default Type Error Message for the Instruction Program


```
[ERROR] #16 Asm: main.asm:14 This error is caused due to wrong instruction name. PLP does not
have an instruction as mentioned in the code.subiu - instruction is not defined in PLP. Unable to
process the instruction.Error occurred around word - "subiu"
Please refer following link for more information http://progressive-learning-platform.github.
io/instructions.html#operations
[ERROR] #24 Asm: preprocess: 1 error(s).
```

Figure 24- Link Type Error Message for the Instruction Program


```
[ERROR] #16 Asm: main.asm:14 This error is caused due to wrong instruction name. PLP does not
have an instruction as mentioned in the code.subiu - instruction is not defined in PLP. Unable to
process the instruction.Error occurred around word - "subiu"

BEFORE CORRECTION
sub $t1, $t2, $t3
AFTER CORRECTION
subu $t1, $t2, $t3
[ERROR] #24 Asm: preprocess: 1 error(s).
```

Figure 25- Example Type Error Message for the Instruction Program

3. Register Program (P3):

This was the third and final program given to the participants during the think aloud experiment. Figure 26 shows the program and its description. The error is at line 28. Here program's intention was to call different subroutines/functions each of them will perform addition, subtraction and multiplication and return their respective operations result. For returning the calculated results, program is using 'v' registers. There are only v0 and v1 register available in PLP. So v2 used in the program is not a register in PLP. Figure 27 gives the error message as provided by the *Default* type, Figure 28 gives the error message as provided by the *Link* type and Figure 29 gives the error message as provided by the *Example* type.

To fix the error in this program, we need to replace $v2 with a register recognized by the PLP. While replacing the register, we should also take care that a new register should be used to avoid overwriting the already used register. For example, we can use $s0 instead of $v2. There are other possible replacement registers also. So, to assemble properly we can replace $v2 with $s0 in line 28. Final instruction after edit would be 'mullo $s0, $a1, $a0'.

```
main.asm <main program>                                    Editor Mode
 1
 2     #This program will perform addition, subtraction and multiplication
       operation on two registers $a0 and $a1.
 3     .org 0x10000000
 4     main:
 5             li $a0, 100
 6             li $a1, 200
 7             jal addition_operation
 8             nop
 9             jal subtraction_operation
10             nop
11             jal multiplication_operation
12             nop
13     end:
14             j end
15             nop
16
17     addition_operation:
18             addu $v0, $a1, $a0
19             jr $ra
20             nop
21
22     subtraction_operation:
23             subu $v1, $a1, $a0
24             jr $ra
25             nop
26
27     multiplication_operation:
28             mullo $v2, $a1, $a0
29             jr $ra
30             nop
31
32
```

Figure 26- Register Program

```
[ERROR] #68 Asm: main.asm:28: Invalid register(s)
[ERROR] #74 Asm: assemble: 1 error(s).
```

Figure 27- Default Type Error Message for the Register Program

```
[ERROR] #68 Asm: main.asm:28 Register used is not recognized by the PLP.mullo is expecting a
register but got something else $v2.Error occurred around word - "$v2"
Please refer following link for more information http://progressive-learning-platform.github.
io/instructions.html#registers-names-and-conventions
[ERROR] #74 Asm: assemble: 1 error(s).
```

Figure 28- Link Type Error Message for the Register Program

```
[ERROR] #68 Asm: main.asm:28 Register used is not recognized by the PLP.mullo is expecting a
register but got something else $v2.Error occurred around word - "$v2"

BEFORE CORRECTION
addu $t1, $t2, $a4
AFTER CORRECTION
addu $t1, $t2, $a3
[ERROR] #74 Asm: assemble: 1 error(s).
```

Figure 29- Example Type Error Message for the Register Program

APPENDIX B

SAMPLE TRANSCRIBING, SEGMENTING, AND CODING

This section will give an example of coded data. The example will be of Participant #28 solving Invalid Label Error Program. For this exercise, error message was displayed using Default type.

Speaker	Audio	Screen	Examine Explore Hypothesize Repair Evaluate
##28##	So first I am going to read the problem and trying to understand what program is doing given two numbers 45 and 60 based on the value in register t1 perform either addition or subtraction operation final result should be stored in register $s1 0 is addition 1 subtraction		Examine
##28##	after assembled I am getting error at main asm 20 unable to process token addition	[ERROR] #16 Asm main.asm:20 unable to process token addition	Examine
##28##	line 20 that would be ahmm it says addition on that line and it appears it could be either a comment		Hypothesize
##28##	I am not understanding exactly what the user was attempting to do since there is an addition below it where in addu operation below it I am going to comment out addition word and see		Hypothesize
##28##	if it should be a jump nope nope since it is a last one		Hypothesize
##28##		##28## comments out line 20. So after edit, line 20 is '#addition'.	Repair
##28##	see if that works save reassemble	Saves the program assembles it. Gets one error. #71 Asm: main.asm:13: Invalid branch target "addition"	Evaluate
##28##	okay so next error says		Examine

64

	asm 13 invalid branch target addition		
	so that based on line of 13 addition is not supposed to be [inaudible word] supposed to be a label so I am adding a colon after the word and		Hypothesize
##28##		##28## uncomments line 20 and adds ':' at the end of the line 20 that is after word 20.	Repair
##28##	I save and compile or assemble again	##28## saves the program. clicks assemble button. No error is generated	Evaluate
##28##	and I do not have any more errors. Do you want me to go and test run the program?		
instructor	Yeah you can its upto you		
##28##	Okay yeah I mean I can test run see if the program works as intended based on the problem I am bring up cpu watcher and organize these screens little bit so I can see cpu watcher and the program and add registers $t1 $t2 $t3 and based on value of based on value of register t1 a an addition step through the program everything looks good jumps to branches to addition does the addition and stores that in $s1 perfect it appears to work so it is not thorough testing but	##28## opens watcher window. Reorganizes the PLPTool window. Adds $t1, $t2, $t3 registers to watcher window. Then starts step simulation. Each instruction is executed stepwise. ##28## adds $s1 register to watcher window.	Examine

Table 17- Sample Coding for Basic Steps

Speaker	Audio	Screen	Expected Gaming T&E Incorrect-Interpretation Incorrect-Independent Correct-Independent Silly Mistake
##28##	So first I am going to read the problem and trying to understand what program is doing given two numbers 45 and 60 based on the value in register t1 perform either addition or subtraction operation final result should be stored in register $s1 0 is addition 1 subtraction		
##28##	after assembled I am getting error at main asm 20 unable to process token addition	[ERROR] #16 Asm main.asm:20 unable to process token addition	
##28##	line 20 that would be ahmm it says addition on that line and it appears it could be either a comment		Incorrect Interpretation
##28##	I am not understanding exactly what the user was attempting to do since there is an addition below it where in addu operation below it I am going to comment out addition word and see		Incorrect Interpretation
##28##	if it should be a jump nope nope since it is a last one		Expected
##28##		##28## comments out line 20. So after edit, line 20 is '#addition'.	Incorrect Interpretation
##28##	see if that works save reassemble	Saves the program assembles it. Gets one error. #71 Asm: main.asm:13: Invalid branch target "addition"	

##28##	okay so next error says asm 13 invalid branch target addition		
	so that based on line of 13 addition is not supposed to be [inaudible word] supposed to be a label so I am adding a colon after the word and		Expected
##28##		##28## uncomments line 20 and adds ':' at the end of the line 20 that is after word 20.	Expected
##28##	I save and compile or assemble again	##28## saves the program. clicks assemble button. No error is generated	
##28##	and I do not have any more errors. Do you want me to go and test run the program?		
instructor	Yeah you can its upto you		
##28##	Okay yeah I mean I can test run see if the program works as intended based on the problem I am bring up cpu watcher and organize these screens little bit so I can see cpu watcher and the program and add registers $t1 $t2 $t3 and based on value of based on value of register t1 a an addition step through the program everything looks good jumps to branches to addition does the addition and stores that in $s1 perfect it appears to work so it is not thorough testing but	##28## opens watcher window. Reorganizes the PLPTool window. Adds $t1, $t2, $t3 registers to watcher window. Then starts step simulation. Each instruction is executed stepwise. ##28## adds $s1 register to watcher window.	

Table 18- Sample Coding for Expected and Unwanted Steps

Speaker	Audio	Screen	Program-

			Description Error-Message Code Search-Information
##28##	So first I am going to read the problem and trying to understand what program is doing given two numbers 45 and 60 based on the value in register t1 perform either addition or subtraction operation final result should be stored in register $s1 0 is addition 1 subtraction		Program Description
##28##	after assembled I am getting error at main asm 20 unable to process token addition	[ERROR] #16 Asm main.asm:20 unable to process token addition	Error Message
##28##	line 20 that would be ahmm it says addition on that line and it appears it could be either a comment		
##28##	I am not understanding exactly what the user was attempting to do since there is an addition below it where in addu operation below it I am going to comment out addition word and see		
##28##	if it should be a jump nope nope since it is a last one		
##28##		##28## comments out line 20. So after edit, line 20 is '#addition'.	
##28##	see if that works save reassemble	Saves the program assembles it. Gets one error. #71 Asm: main.asm:13: Invalid branch target "addition"	
##28##	okay so next error says asm 13 invalid branch target addition		Error Message

	so that based on line of 13 addition is not supposed to be [inaudible word] supposed to be a label so I am adding a colon after the word and		
##28##		##28## uncomments line 20 and adds ':' at the end of the line 20 that is after word 20.	
##28##	I save and compile or assemble again	##28## saves the program. clicks assemble button. No error is generated	
##28##	and I do not have any more errors. Do you want me to go and test run the program?		
instructor	Yeah you can its upto you		
##28##	Okay yeah I mean I can test run see if the program works as intended based on the problem I am bring up cpu watcher and organize these screens little bit so I can see cpu watcher and the program and add registers $t1 $t2 $t3 and based on value of based on value of register t1 a an addition step through the program everything looks good jumps to branches to addition does the addition and stores that in $s1 perfect it appears to work so it is not thorough testing but	##28## opens watcher window. Reorganizes the PLPTool window. Adds $t1, $t2, $t3 registers to watcher window. Then starts step simulation. Each instruction is executed stepwise. ##28## adds $s1 register to watcher window.	

Table 19- Sample Coding for Examine Steps

Speaker	Audio	Screen	Complete Partial

69

			Ignore
##28##	So first I am going to read the problem and trying to understand what program is doing given two numbers 45 and 60 based on the value in register t1 perform either addition or subtraction operation final result should be stored in register $s1 0 is addition 1 subtraction		
##28##	after assembled I am getting error at main asm 20 unable to process token addition	[ERROR] #16 Asm main.asm:20 unable to process token addition	Complete
##28##	line 20 that would be ahmm it says addition on that line and it appears it could be either a comment		
##28##	I am not understanding exactly what the user was attempting to do since there is an addition below it where in addu operation below it I am going to comment out addition word and see		
##28##	if it should be a jump nope nope since it is a last one		
##28##		##28## comments out line 20. So after edit, line 20 is '#addition'.	
##28##	see if that works save reassemble	Saves the program assembles it. Gets one error. #71 Asm: main.asm:13: Invalid branch target "addition"	
##28##	okay so next error says asm 13 invalid branch target addition		Complete
	so that based on line of 13 addition is not supposed to be [inaudible word] supposed to be a label so I		

	am adding a colon after the word and		
##28##		##28## uncomments line 20 and adds ':' at the end of the line 20 that is after word 20.	
##28##	I save and compile or assemble again	##28## saves the program. clicks assemble button. No error is generated	
##28##	and I do not have any more errors. Do you want me to go and test run the program?		
instructor	Yeah you can its upto you		
##28##	Okay yeah I mean I can test run see if the program works as intended based on the problem I am bring up cpu watcher and organize these screens little bit so I can see cpu watcher and the program and add registers $t1 $t2 $t3 and based on value of based on value of register t1 a an addition step through the program everything looks good jumps to branches to addition does the addition and stores that in $s1 perfect it appears to work so it is not thorough testing but	##28## opens watcher window. Reorganizes the PLPTool window. Adds $t1, $t2, $t3 registers to watcher window. Then starts step simulation. Each instruction is executed stepwise. ##28## adds $s1 register to watcher window.	

Table 20- Sample Coding for Complete, Partial, Ignore

APPENDIX C

Consent and Participant Recruitment Form

Impacts of error messages on student's ability to understand and fix errors in programs

I am a graduate student under the direction of Dr. Scotty Craig in the Department of Engineering at Arizona State University. I am conducting a research study to examine the impact of different forms of error messages in PLP (Progressive Learning Platform) language on student's ability to understand and fix errors in programs. Please be assured that your responses will be kept completely confidential.

The study should take you around 1 hour to complete, and you will receive $10 for your participation. Your participation in this research is voluntary. You have the right to withdraw at any point during the study, for any reason, and without any prejudice. If you have questions, concerns, or complaints, contact Dr. Scoty Craig at (xxx) xxx-xxxx or scotty.craig@asu.edu. If you have any questions about your rights as a subject/participant in this research, or if you feel you have been placed at risk, you can contact the Chair of the Human Subjects Institutional Review Board, through the ASU Office of Research Integrity and Assurance, at (480) 965-6788.

I am inviting your participation, which will involve demographic survey, think aloud programming exercise consisting of three PLP (Progressive Learning Platform) programs and a short interview about the three PLP programs. You have the right to refuse to answer any of the questions, and to stop participation at any time. There are no foreseeable risks or discomforts to your participation.

If you have previously taken SER 250/SER 520 course or currently taking SER 250/SER 520 course, your grades in SER 250/SER 520 may be used for data analysis.

This will not have any impact on your grades in SER 250/SER 520. Your name and other

identifiable information will only be used during the study for proper data analysis. They

will be removed in the final results. Results will only be shared in the aggregate form.

The results of this study may be used in reports, presentations, or publications but your

name or other identifiable information will not be used.

Everything visible to you on the computer monitor will be recorded (commonly

referred to as screen recording). The entire session will be audio recorded. The screen

audio recording will not take place without your permission. Please let me know if you do

not want the screen or audio to be recorded; you also can change your mind after the

recording starts, just let me know.

By clicking the button below, you acknowledge that your participation in the

study is voluntary, you are 18 years of age, and that you are aware that you may choose to

terminate your participation in the study at any time and for any reason.