Checkpoint 1: CIS*4650

Christopher Katsaras & Ryan Lafferty

What has been done?

The first thing that we did was go through the CMinus spec and identify all of the tokens that needed to be created. After Ryan had identified and implemented the tokens required, he then read through the spec writing down the grammars required for the assignment. We then worked together in order to create a set of objects for the grammars written based on the tiny objects that were given to us. The tiny objects provided by Prof. Song helped us gain insight into how to effectively organize our object hierarchy. When making the objects, Chris began by making simple ones like TypeSpec.java and IntExp.java. After establishing these objects, both Ryan and Chris were able to construct the higher level objects and implement and inheritance based system, similar to Prof. Song's. After the objects were implemented we were able to update the Absy.java file in a bottom up fashion in order to produce the required parse trees. We used the tiny code as a basis for the construction of our Absyn.java file. Throughout the entire process of creating and implementing the objects in a bottom up fashion we had to debug all issues that were present in our grammars and objects. Ryan proved to be an excellent debugger when it came to the obscure errors that were produced by CUP. He managed to debug nearly all bugs in a very timely manner. We were able to understand the codebase of the tiny example with little issues which allowed us to design our implementation in a similar fashion. Creating and implementing the objects in a bottom up fashion allowed us to procedurally test the objects and parse tree's as we developed them. Chris proved to be an effective tester as he spent time testing the objects in an effective manner ensuring maximum coverage. After a few days of

development we were able to get all of the objects and parse trees working. Since our implementation was similar to the provided tiny implementation we were able to get syntax error recovery for free, using the same error reporting and recovery method. However, our error messages are more distinct which helps users easily understand where they have made a mistake in their syntax or logic.

Timeline

We will describe briefly the timeline of development for checkpoint one in point form.

Read CMinus Spec - 1 Day

Write Tokens - 1 Day

Write Grammars - 2 Days

Write and Implement Objects - 4 Days

Testing Parse Trees - 2 Days

Testing Error Recovery - 1 Day

Write Up - 2 Days

Total Time - 13 Days

Design Process

When it came to designing our project, we wanted to ensure that we were following the general structure laid out by Prof. Song. For this reason, our grammar was nearly identical to the CMinus specification. However, we had to ensure that we added precedence into our implementation. This ensured that our grammar was free from ambiguity. From there, it was simply a matter of implementing our objects in an effective hierarchical system, similar to Prof. Song's. When it

came to designing our error detection, we decided to take a high level approach and catch errors at higher level rules e.g Dec and Expr. Overall, the design process was interesting and somewhat challenging at times but we felt that in the end we were effectively able to implement an elegant soltuion.

Lessons Gained

During this project, both of us were able to learn a great number of things. CUP is a tool that neither of us had any prior knowledge or experience with and checkpoint one gave us an opportunity to understand how to use it. Both of us had previously used tools like Lex and Yacc but now, with the addition of CUP, we were able to implement a much more sophisticated and elegant system to deal with not only scanning but also building our syntax tree. In regards to building syntax trees, this was a relatively new concept and neither of us had fully implemented a system where we could interpret a language like C minus. Thanks to the help of Prof. Song's sample code, we were able to implement a scanner and parser for C minus which allowed us to translate the concepts we learned in class.

We both felt that one of the biggest benefits of this project was that it allowed us to apply the concepts that we learned in class to a real-world example. Over the past couple months we have gone into great detail with Prof. Song about concepts like scanning, parsing, etc and we felt that this project help us to gain a greater understanding due to the fact that we were able to deal with it, hands on.

Assumptions and Limitations

Similar to C, unbalanced parenthesis can cause undetectable errors. For instance:

```
int main (void) {
        }{
}
```

Will cause undetectable errors due to the fact that the opening brace doesn't know which closing brace to match with.

For this reason, we assume that these kind of unbalanced parenthesis issues will not be introduced into any .cm files that are tested by our teaching assistant Dave Wickland. If for any reason he decides to test this scenario, our implementation will give undesirable output.

Furthermore, subsequent errors will possibly lead to globbed together error statements because of the level that errors are evaluated at.

e.g

rick

adads

(Q)(#$)(!@#)$

int main (void) {

}

Possible Improvements

Due to time constraints, we were unable to make our error messages as specific as we would have liked. If we had more time, we would have ideally implemented lower lever error messages which would be far more specific. By adding these detailed messages, users would be able to

easily identify errors. An example of one of these errors would be unmatched parentheses and braces.