F29LP LANGUAGE PROCESSORS

Coursework 1- Front End

func Programming Language

Ali Muzaffar Ahmad Solehria
H00206125

This part of the coursework was about building the front end of the programming language func. The specification and grammar for this language was given.

The coursework was divided into three parts:

1. **Flex file with suitable token representations**
2. **Recursive Descent Parser**
3. **Abstract syntax tree**

Flex File

With the help of the slides and sample code for “simp” language, this part was started. Lexical analysis was done.

After the lexical analysis was carried out, tokens.h file was made.

This file includes all the tokens that are to be used when programming in func language. Many of the tokens were different to that in the simp language so, a new file was made from scratch to accommodate all the tokens of func language.

After the tokens.h file, the .flex was made. This file included user definitions, lex definitions, rules and user functions. All of these parts were populated according to the func language specification.

Next print.c was also made to give appropriate output to each token.

This concluded the first part of the front end. This was the easiest part of all. Next followed the recursive descent parser.

Recursive Descent Parser

This part started of with factoring func grammar. Each grammar specification was checked if it required factoring, and wherever needed it was carried out.

After factoring the parser was to be written. The slides proved to be really helpful for this part. They gave a good explanation on how to go about things.

Though they look very similar, but simp language has some very fundamental differences from func language.

In func each command needs to end with a semi colon where as in simp only if there is another command following the preceding command, will a semi colon be present. Due to these differences, understanding of the lex function and how a recursive descent completely works was paramount.

The most difficult part of the recursive descent parser was the exprs and expr. Though it proved quite challenging, after time spent on it, it was completed.

The binary operators in the specification were given, and accordingly implemented. Four more functions were also provided in the specification:

1. **Plus**
2. **Minus**
3. **Times**
4. **Divide**

These functions also only expected two expressions. Therefore these were also parsed through the function bop() as the other binary operators.

The specification also specified that the “**Main**” function would not include any arguments and will not return any value. The parser also checks this and gives an error if the **Main** function has any argument or returns a variable.

In the initial stages the parser was tested with a small piece of code. Problems involving lex arrived. Example: lex was called twice in two different functions for same purpose. These problems were resolved and the recursive descent parser was tested with all the examples provided in the pdf.

Programs with deliberate errors were also tested with the parser to check if the parser gave any errors.

To conclude this part, the recursive descent parser was built to support every little detail given in the specification file. Whether it be that some tokens only support binary operations or main function doesn’t have any argument and returns nothing, everything was covered.

Abstract Syntax Tree

Once the recursive descent parser was written, the abstract syntax tree was written. Before the abstract syntax tree was developed, three changes were to be made to the grammar given:

1. **Factor out singleton rules**
2. **Drop unnecessary terminal symbols**
3. **Delete singleton recursion**

These three processes were carried out. Next part was to write the abstract syntax tree.

Each node would have a tag of its own, these were decided for each node. The hardest part was to write for the function. Each function had multiple trees, so to incorporate all of them required very good understanding of the topic.

The slides and the sample code proved to be very effective in understanding the concept completely.

As the recursive descent parser was developed completely, building an abstract syntax tree didn't prove to be as difficult as initially thought.

Some segmentation faults were encountered at the start, but as the understanding of the concept improved, these faults were also fixed.

As with the recursive descent parser, the abstract syntax tree was tested with all the example programs, and each one ran successfully.

Conclusion

Building a compiler from scratch, sounds and is very exciting. This proved to be true during this coursework. I was able to complete all parts of the coursework and tested with all the examples given in the pdf.

The source code is in the folder "Source Code". Inside this folder are three files:

1. func.lex
2. tokens.h
3. print.c

This folder also contains two folders:

1. Recursive descent parser

   This folder contains file "parse.c" which is the recursive descent parser.
2. Abstract Syntax Tree

   This folder contains file "parse.c" abstract Syntax Tree.

Both the recursive descent parser and abstract syntax tree are named parse.c, so if the files are to be compiled, the MakeFile provided can be used without changing the contents of the MakeFile. Because both files are named parse.c, they are put in different folders.