COMP 520 Milestone 1

Rajveer Gandhi, Archit Agnihotri, Dipanjan Dutta
260719747, 260777127, 260779557

Abstract

This is a design and implementation report for Milestone 1 of the GoLite project. This project was implemented using flex and bison.

---

1. Team member contributions

Dipanjan Dutta

- Created the template for scanner and parser
- Created the example valid (6) and invalid (30) programs
- Implemented the weeding phase

Rajveer Gandhi

- Implemented and debugged the parser
- worked jointly with Archit on the AST & pretty printing

Archit Agnihotri

- Implemented and debugged the scanner
- Worked jointly with Raj on the AST & pretty printing

All of us

- Decided on the language for design as C
- Discussed shift/reduce conflicts arising in the parser and scanner bugs and resolved them on discussion
- Everyone made their individual contribution to this document

2. Issues Faced

Dipanjan started off by creating a preliminary sketch of scanner and parser. Raj and Archit then worked on finding errors and shift/reduce conflicts in the grammar. Some of the problems raised involved a group discussion over conference calls, which always ended on an unanimous decision to an approach of solving the problem. The problems faced were :

2.1. Optional semicolons

We had to deal with this by creating a variable called *lastRead*, and updating it whenever a new token is generated (before it is returned). Then, when $\backslash n$ or **EOF** or block comment including a $\backslash n$ is encountered we check to see with function *insertSemicolon* if there should have been a semicolon there, and we insert the semicolon in that case.

2.2. block comments

We encountered a bug where we were not inserting semicolons after block comments all the time. Upon debugging, we realized that we needed to split the regex for block comments into one that include newlines and ones that don't.

2.3. Tokens mode

We use a global variable in ***main.c*** called *g_tokens* (g for global) and if it is turned on then we print the token value so that we can print the tokens for the compiler mode "tokens"

2.4. Escape sequence handling

This was perhaps the most challenging thing that we had to face in the scanner and Archit handled it. Archit and Raj created the regular expressions such that only the supporting escape sequences are valid.

2.5. Block comments

Dipanjan initially added a state machine implementation of the handling of block comments. However, it was too abstract and none of us knew how exactly it was handling the block comments. So we later decided to switch to regular expressions.

Once all of us were satisfied with the performance of the scanner and parser, Raj and Archit moved on to create the AST while Dipanjan started working on the report and also created 30 invalid and 6 valid programs to submit with the code.

2.6. Operator Precedence and Associativity

Operator precedence and associativity was handled by using the precedence declaration: %left symbols

2.7. Shift/Reduce Problems

We encountered a few shift/reduce errors in particular with how idlist and other_expressions were handled. These issues were resolved by rewriting rules

pertaining to idlist and other_expression to make the grammer LALR(1). We were able to solve the common dangling else problem by always using opening and closing curly braces around the statement.

2.8. AST

A problem we had in implementing the AST was dealing with the multiple recursive elements that are not disjoint as given in the specifications. In particular, we have three types of top level declarations (var, type and function). However, only var and type can occur within statements. Both "top level declarations", "expressions" and "statements" are recursive. We handled this problem effectively by creating the NODE struct in such a way that linked lists were handled correctly. We organized it by having a "NODE *next" field in each part of the struct that we needed to create a linked linked separately for statements, declarations and expressions.

2.9. Pretty Printer

In addition to the sheer magnitude of the AST that is required for traversal during pretty printing, we encountered a few issues that were caused by non-optimal design of the AST. We then went back to the AST and redesigned it. The successful output of the pretty output confirmed that we had created the AST correctly. We also use a global variable g_indent to calculate the indentation we must have to make our output "pretty". We decided not to use tabs, and use 4 spaces as one level of indentation.

2.10. Weeding

Because some errors cannot be caught in the parsing stage (as it is allowed by the grammar), we implemented a separate weeding phase to eliminate

those errors. The errors include :

- Ensuring that there is one default in a switch block - We used a static flag hasDefault (initially set as false). Whenever we are in a switch block and we have encountered default, we turn this flag to true. If we encounter a default with the flag being true, we throw an error and exit the program

- Ensuring break and continue are used legally: We used two static flags - insideFor and insideSwitch(both initially false). If either of the flags are false and we encounter break statement, we throw an error. If we see that the insideFor is false and we encounter continue statement, we throw an error.

- Blank identifiers: We use a static flag isBlankIdValid (initially false). If we encounter an id list for top-level declaration or assignment statements, we set the flag as true and recurse on the idles and LHS of assignment respectively. If we encounter an identifier with the flag as true, we throw an error.