Andrew Crowell
Andrew Judd

CIS*4650
Pish Compiler
Term Project – Checkpoint #1

What was completed?

The first checkpoint for the term project can be separated into two main components; grammar derivation and abstract syntax tree creation.

Grammar Derivation

Based on the examples and the predefined grammar, we manipulated the grammar in order to incorporate all of the undefined functionality of the language into it. In doing so, we also introduced numerous new states and added transitions between them, so that the language is able to be recognized and parsed as required for the later parts of the compiler.

Abstract Syntax Tree Creation

After the parser was complete, the creation of the abstract syntax tree was started. Since the base language for the application is C++, we used object-oriented programming styles in order to create the entire tree. This allowed us to define the base class which all of the other objects extend. Within the base class, a virtual function was also provided in order to allow for the definition of class specific dump functions which will work with the '-a' flag.

Compiler Backend

Along with the derivation of the grammar, and the creation of the abstract syntax tree, other core functionalities for the compiler were created. For example, the pishc application will accept two of the compiler flags which are required for the rest of the application. Currently there are two compile types available (more will be added in later as need be). They are:

- -a – this will create an output file '<filename>.abs' which contains the dumped version of the abstract syntax tree
- All other flags will compile the code to the point where we are able to either accept or reject the input provided. It will finish by telling whether the provided data file passed or failed.

Error Recovery

Since there are lots of odd quirks about the syntax of the language, it is extremely easy for errors to arise in any code which is written for it. Because of this, error recovery is a must. By leveraging the built-in error features in Bison (the GNU distribution of YACC), we were able to capture any errors which may arise and then jump forward in order to continue the processing of the input file. However, this is not always the case. In some cases, recovery is impossible since the parser will enter an inconsistent state in which it is impossible to escape from. These are usually caused by parse errors while handling the more bizarre syntactic elements of the language. Otherwise, our compiler will allow a maximum of 15 error messages to be raised before it decides that there are too many errors to continue and then terminates. If required later on, this 15 is able to be raised and/or lowered in order to suit the needs for the future.

Abstract Syntax Tree

As previously mentioned, the abstract syntax tree is able to provide a visual representation of the nodes which have been stored inside of it. For an example of the visual representation of the abstract syntax tree please refer to Figure 1.

The indentations of each line in the .abs file represent the depth in the abstract syntax tree. For example, the 'head' expressions (line 2 of Figure 1) is deeper in the tree than 'Program', but higher in the tree than the 'type', 'name', and 'identifierList' nodes. In general, the deeper the element is in the tree, the more recent it was created by the parser. So during construction, a particular node must have its subtree filled before it is created. In contrast, iteration on the semantic validation level will start at the highest depth ('program'), and work down to the leaf nodes of the AST.

```
6: Program(
    head = 2: ProgramHead(
        type = HEAD_MAIN, name = 2: Identifier(name = TestEmptyProgram),
        identifierList = 2: IdentifierList(
            2: Identifier(name = input),
            2: Identifier(name = output)
        )
    ),
    body = 6: ProgramBody(
        locals = 2: DeclarationList(),
        statements = 6: StatementList()
    )
))
```

Figure 1 - Sample Abstract Syntax Tree Output (line numbers are shown in front of nodes)

Implementation Details and Assumptions

Due to some inconsistencies which lie within the grammar as well as some very restrictive specifications for the language, some assumptions were made in order to relax the language slightly, and to correct behaviours in a few ambiguous parts of the grammar. The notable implementation details and assumptions are as follows:

- Type alias sections, variable sections and constant sections can appear in any order – we will use semantic analysis in order to verify that each of these are placed in the correct symbol tables in the future. However, for the sake of writing test cases compatible with other people's compilers, we assumed that it was the more strict order as specified in the language specifications. This made certain aspects of the parser and AST generation simpler.
- No variable return types are validated until semantic checks – This will allow for the type aliases for standard ones to be returned as well (since they will still use the same storage as standard types).

- String literals are expressions in single quotes that can be assigned to an array of characters.
- Identifiers are allowed to have underscores, and are permitted in the same positions as letters are allowed in identifiers.
- Our parser supports as many features as possible as defined in the specification: records, arrays, constants, nested type expressions (used by variable and type alias declarations), nested subprograms, forward declarations, control structures, statements, subprogram calls, comments, and so forth.
- Our parser will recover from errors if it is possible to do so. This is described earlier.

Contributions

Currently the majority of the work for this project has been split evenly between both members of the team. Given that most of the work from the beginning until now is fairly linear in progress (i.e. parallel programming at this point in time would be rather difficult), we worked together in the development of all of the components. Thus our task lists are identical here:

Andrew Crowell	**Andrew Judd**
▪ Evaluation of pros and cons of potential frameworks and languages to use ▪ Grammar design speculation ▪ Actual grammar implementation ▪ Designing the object-oriented abstract syntax tree class hierarchy ▪ Building the abstract syntax tree ▪ Testing the abstract syntax tree ▪ Correcting bugs at various levels (lexer, parser, AST, compiler front-end) ▪ Development of adequate test cases ▪ Checkpoint #1 documentation	▪ Evaluation of pros and cons of potential frameworks and languages to use ▪ Grammar design speculation ▪ Actual grammar implementation ▪ Designing the abstract object-oriented syntax tree class hierarchy ▪ Building the abstract syntax tree ▪ Testing the abstract syntax tree ▪ Correcting bugs at various levels (lexer, parser, AST, compiler front-end) ▪ Development of adequate test cases ▪ Checkpoint #1 documentation