Andrew Crowell
Andrew Judd

CIS*4650
Pish Compiler
Term Project – Checkpoint #2

What was completed?

The second checkpoint for the term project consisted of one main area of validation; semantic analysis and the creation of the symbol table. On top of these components, a few minor fixes were applied to the grammar when issues arose.

Grammar Patches

Upon further inspection, we noticed that there were a few holes in the grammar which either lead to states which made the semantic analysis more difficult than needed. As well as the minor issues which allowed for further analysis in the grammar, one major issue was found within our original grammar which needed to be fixed. This issue was caused by escape characters being stored in string literals.

Semantic Analysis and Symbol Table Creation

After the abstract syntax tree has been created, the process of semantic analysis begins. During this process we build up two symbol tables as we go in order ensure that everything which has been used throughout the application has been defined as well as validating the types of data which goes along with them.

Compiler Backend

Since more options are now available in the compiler, more of the compiler flags have been activated. There are three modes which have been enabled in our compiler. They are:

- -a – this will create an output file '<filename>.abs' which contains the dumped version of the abstract syntax tree
- -s – this will create an output file '<filename>.sym' which contains the dumped version of the symbol table.
- All other flags will compile the code to the point where we are able to accept or reject the input provided based on both syntactic and semantic rules

Symbol Table

The symbol table which is being created for the purposes of type checking and data validation is able to be displayed. Since the main use of this is part of the application is to verify that data is being

captured as the correct types, the output from this function is relatively verbose. This has been proven as beneficial throughout the testing of the application we were trying to determine why the semantic analysis kept failing, it wasn't until the output from the symbol table was printed that a mistake in the code was noticed. Sample output from the symbol table can be seen in Figure 1.

```
GLOBAL SYMBOL TABLE (VARIABLES, CONSTANTS, SUBPROGRAMS):
    a : variable of type integer
    b : variable of type integer
    false : constant of type INTEGER (0)
    input : main program parameter
    maxint : constant of type INTEGER (2147483647)
    output : main program parameter
    testforloop : self (used for return value assignment, and self-
recursion)
    true : constant of type INTEGER (1)
    writechar : built-in procedure writechar
    writeint : built-in procedure writeint
    writereal : built-in procedure writereal
    x : variable of type integer

GLOBAL TYPE TABLE:
    char : primitive type char
    integer : primitive type integer
    real : primitive type real
```

Figure 1 - Sample Symbol Table Output

Implementation Details and Assumptions

In order to provide a slightly more lenient compiler, several design decisions were made which has the potential for allowing for variability in the code written for it. Despite these changes, our tests do not reflect the more lenient definitions so that other groups will be able to use them as tests for their applications. The notable implementation details and assumptions are as follows:

- 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.
- Variable, constant, type, and subprogram declarations can appear in any order and be mixed (so long as they refer only to things declared above them), and constants and types can locally-scoped to nested functions.
- 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.

- When validating arrays, an array expression of less than or equal to the left-hand side's size of an array is acceptable (i.e. an array of size 2 is assignable to an array size 4). Another side-effect of this is arrays of the same size but with different boundaries are assignment-compatible with one another.
- Subprogram forward declarations are more like C, and expect you to still define the entire formal parameter list again when you fill in the body for a function for later.
- Temporary variables are generated for any expressions that are not a single term, when passed into a function call. This allows for potentially strange, but valid behaviour like swap(a, b + c / 4).
- String literals are terminated by null characters when assigned to array of char, and take length + 1 characters.
- String literals assigned to char variables must be exactly length 1 (because char can only hold exactly one char).
- Comparison is allowed between ANY two valid expressions, but if they are not type-compatible, the emitted code will result in 0 being returned. This is necessary to compare integers to reals, chars to integers, etc.
- If there are syntax errors, the semantic checking phase does not occur.

Contributions

For the most part the project still needed to be completed fairly linearly. For that, both members of the team have been actively participating in the development process. For the parts we were able to start parallelizing (i.e. parts within the type checking) we split the required functionality between the two of us and completed them.

Andrew Crowell	**Andrew Judd**
▪ Detected and corrected bugs in the grammar ▪ Type checking and semantic analysis ▪ Generation of the symbol table ▪ Development of adequate test cases ▪ Checkpoint #2 documentation	▪ Detected and corrected bugs in the grammar ▪ Type checking and semantic analysis ▪ Generation of the symbol table ▪ Development of adequate test cases ▪ Checkpoint #2 documentation