1. Introduction

The third phase of snuplc is semantic analysis. The basic requirement was to add support for type checking, type inference, and expression evaluation to the parser built in the second phase, and to treat symbolic constants as previously evaluated constants. In addition, support for parsing minimum possible integer value in 32-bit system was implemented in the form of 'relaxed' parsing of unary negation operation '-'.

2. Basic Implementation

As mentioned above, newly implemented parser needed to support type checking, evaluation, and substitution of symbolic constants with appropriate constants. This section will provide explanations for the corresponding implementations. Implementations were mostly completed without altering the structure of the code from the second phase, with some exceptions where additional arguments had to be passed to some functions, or the return type of some functions needed to be changed.

2.1. Type Checking

Type checking is implemented in member functions of the abstract syntax tree nodes. The function is named TypeCheck. All type checking functions must satisfy the following property. For each node that type checks, the TypeCheck function returns true if and only if there is no type error in the entire subtree whose root is that node. Each TypeCheck function accepts as arguments a pointer to a CToken instance and a pointer to a string for error reporting purposes. When type error is detected, the token of the abstract syntax tree node that is performing the type check and a relevant error message is written to the address that is pointed by the arguments.

Type checking is invoked on the root node of the abstract syntax tree, the CAstModule instance, after parsing is completed. By the property of the type checking function explained above, this should ensure that all nodes in the abstract syntax tree contain correct types.

Before inferring the type of a node or evaluating the value of a node, type checking must be preceded. Otherwise the inferred type or evaluated value is not guaranteed to be accurate. Sometimes inferring types of children nodes is required to type check some node. In such a case children nodes are always type checked before the inference occurs. Evaluation is required in the parsing stage for acquiring correct array lengths for array declarations. This too requires early type checking before the global type check that is going to occur after parsing is completed.

2.2. Type Inference

Type inference is implemented in member functions of the abstract syntax tree nodes. The function is named GetType. Type inference of a node presumes that the subtree that has the node as root has been type checked, and no incorrect type is contained in all the descendant nodes. This is achieved by always ensuring that TypeCheck is called before GetType is called. TypeCheck ensures that all descendant nodes contain correct types.

2.3. Evaluation

Evaluation is implemented in member functions of the abstract syntax tree nodes that inherit CAstExpression class. The function is named Evaluate. Evaluation of a node presumes that the subtree that has the node as root is TypeChecked. Thus, a node's TypeCheck must be called before calling Evaluate. Evaluation returns an instance of CDataInitializer. CDataInitializer is CDataInitInteger, CDataInitBoolean, CDataInitChar, or CDataInitString. The instance contains appropriate data. Evaluation of a parent node is inherently dependent on the evaluation of its children nodes. For example, if we are to evaluate a binary integer summation, we would need the values of the left-hand-side and right-hand-side to compute the sum. The evaluation is implemented so that the sum becomes a new value instance, to avoid having to make the strange decision of whether to update left-hand-side CDataInitializer or right-hand-side CDataInitializer and delete the other one. Instead, each evaluation frees CDataInitializers from its children's evaluation and returns a new instance. The exception is that the CDataInitializers contained in the symbol table should not be freed. So, the evaluation of CAstDesignator just returns a new CDataInitializer with the same internal value of its symbol's CDataInitializer.

2.4. Symbolic Constants

In the abstract syntax tree, symbolic constants need to be treated as a constant node. That is, we should be able to replace or symbolic constants with its assigned values at compile time. Otherwise little justification exists for having such feature as part of the language. For symbolic constants of scalar types, the symbolic constants are parsed as CAstConstant, with the value that is stored in the symbol representing the symbolic constant. For symbolic constants of string type, the symbolic constants are parsed as a CAstDesignator that, instead of pointing to the symbol of the symbolic constant itself, points to the global string variable that was added to the symbol table when parsing the declarations of symbolic constant.

3. Additional Implementation

3.1. Relaxed Folding of Unary Negation

We chose to implement the relaxed folding of unary negation. For the implementation, the parser's functions were modified to accept an extra boolean parameter, fold. The modified functions are term and factor, which are called from simpleexpr, where the unary operation is first met. If the unary operation of opNeg is recognized, then simpleexpr looks ahead 1 token. If the peeked token is tNumber or tLParens, then fold is set to true, and term is called with fold and passes the value of fold to factor. If factor sees a number, then the factor parses a CAstConstant that contains the negative of the CAstConstant it gets from calling number. If factor sees tLParens, then the factor parses a CAstUnaryOp between the opNeg and the parenthesized expression. If simpleexpr does not see a number or a tLParens, then simpleexpr returns a unary operation between opNeg and whatever is to be parsed by calling term.

4. Conclusion

By the third phase of the project we were able to exactly match the output of the reference binary file. All given files from the second and third phases were tested, and the outputs showed exact

matches when the input file was syntactically and semantically correct. Differences between outputs persisted when the input file was syntactically or semantically incorrect. This was due to the differences in error messages. Even in erroneous conditions, the token's position of the error showed exact match.

Some errors from the second phase was discovered and fixed. For example, the parser from the second phase treated type declarations in symbolic constant declaration and formal parameters as same. This resulted in the arrays declared in symbolic constant declaration as being a pointer to an array. Due to the same cause, symbolic constant declaration allowed all types of arrays. This was fixed by adding an additional boolean parameter to type declaration parsing function to indicate whether we are parsing formal parameters or symbolic constant declaration. Also, a minor bug where null tokens were passed to abstract syntax tree nodes were fixed.