Source to Source Compiler (SIL to C++)
Roll No : 112001010

Makefile

`make test`

create AST and C++ file from test.txt file *present in current directory*.
to run generated c++ file use,

```
g++ test.cpp -o test.out && ./test.out
```

`make tests`

create AST and C++ file from all testcases *present in directory testcase*
create AST and C++ code in *AST_output and CPP_output* directory respectively.
to run generated c++ file use,

```
g++ [./CPP_output/{filename}] -o {filename}.out && ./{filename}.out
```

`make clean`

delete all generated files.

---

compiler.y

This is a yacc/bison file for a compiler. It defines the grammar for a programming language and generates a parse tree.

The grammar specifies the syntax rules of the language, such as the order and structure of statements and expressions. The parse tree is a hierarchical representation of the program, where each node represents a statement or expression and its children represent its operands.

The grammar includes rules for global declarations, function definitions, statements such as conditional statements and loops, and expressions such as arithmetic and logical operations. The parse tree generated by this grammar will be used by later stages of the compiler to generate executable code.

In addition to the grammar rules, this file also includes C++ code fragments that are executed during parsing. These code fragments can be used to perform actions such as constructing nodes of the parse tree or checking for syntax errors.

Compiler.l

The purpose of this code is to define the language's keywords, operators, and other tokens that can be recognized by the lexical analyzer.

Each keyword or operator is associated with a token, which is returned to the parser when encountered. For example, the keyword "main" is associated with the token MAIN, and the operator "==" is associated with the token EQUALEQUAL.

The code also includes regular expressions for recognizing numerical values and variable names. When a numerical value is recognized, it is converted to an integer and stored in a node of the abstract syntax tree, which is then returned to the parser with the NUM token. When a variable name is recognized, it is stored as a string in a node of the abstract syntax tree, which is then returned to the parser with the VAR token.

Overall, this code defines the lexicon of the SIL programming language and is used to tokenize input source code for further processing by the parser.

AbsSynTree.hpp

The given code is a C++ program that defines a data structure called "node" and some functions to manipulate it. The "node" structure represents a node in an abstract syntax tree, which is commonly used in programming language compilers and interpreters.

The program uses various constructors to initialize the "node" structure with different properties, such as storing data types, variable names, and function definitions. It also has functions to add nodes to the tree, which are used to build the tree from its individual nodes.

The program also includes a function called "parse" which takes three "node" parameters and performs some operation on them. The purpose and implementation of this function are not clear from the given code.

AbsSynTree.cpp

The "info" struct contains a string "name" that represents the name of the variable. It also contains an integer "val" that represents the value of the variable, an integer "type" that

represents the type of the variable (e.g., integer, float, etc.), an integer array "arr" that represents an array variable (if the variable is not an array, arr is null), and an integer "size_" that represents the size of the array (if the variable is not an array, size_ is -1). The struct also contains a map called "symboltable" that represents a symbol table for the variables.

The constructor for the "info" struct takes three arguments: a string "s" that represents the name of the variable, an integer "type" that represents the type of the variable, and an integer "size_" that represents the size of the array (if the variable is not an array, size_ is -1). If the type is 2 (i.e., an array), the constructor initializes the "arr" array to the specified size. If the type is 3 (i.e., a function), the constructor sets the "type" member variable to 3. There is also an overloaded constructor that takes an additional integer "datatype" argument, which represents the data type of the variable.

The getNew function takes a pointer to a character array p and creates a new character array temp with a size of 100. It then copies the contents of p to temp until it reaches the end of p. It adds a null character at the end of temp and returns the temp array.

The printExpr function takes a pointer to a node structure representing the root of the AST. The function recursively traverses the AST using a series of if statements to determine the type of node and the corresponding operation. The AST is printed to the ast output stream and the corresponding C++ code is printed to the cpp output stream.

If the root node is NULL, the function returns. If the root node is a variable node, the function checks if it is a variable or an array reference and prints the corresponding information to ast and cpp.

If the root node represents a function call, the function prints the function call information to ast and cpp. It then recursively prints the argument list to cpp.

If the root node is a number node, the function prints the number to ast and cpp.

If the root node represents a binary operator such as addition, subtraction, multiplication, or division, the function recursively prints the left and right operands to ast and cpp with the corresponding operator.

If the root node represents a comparison operator such as less than, greater than, equal to, or not equal to, the function recursively prints the left and right operands to ast and cpp with the corresponding operator.

If the root node represents a logical operator such as AND, OR, or NOT, the function recursively prints the left and right operands to ast and cpp with the corresponding operator.

Finally, if the root node represents a boolean value, the function prints the boolean value to ast and cpp.

Overall, this code snippet provides a useful tool for generating C++ code from an AST representation of an expression.

The code appears to be implementing a function called ASTStatements that takes a pointer to a node as input. This function recursively traverses the syntax tree represented by the node structure and generates C++ code and an abstract syntax tree (AST) representation of the input program.

The function checks the type of each node in the syntax tree using the type field of the node structure. Depending on the type of the node, it generates the appropriate code and AST representation.

For instance, if the type field is 507, the function generates code and AST for an assignment statement. If the type field is 512, the function generates code and AST for a while loop. If the type field is 505, the function generates code and AST for a read statement. If the type field is 506, the function generates code and AST for a write statement. If the type field is 511 or 510, the function generates code and AST for an if-else statement.

The function also handles function calls (type 600) by generating code and AST for the function call and its parameters.

In summary, the ASTStatements function is part of a larger program that generates C++ code and an AST representation of a given input program. It appears to handle most of the major language constructs such as assignments, loops, conditionals, and function calls.
The parse function in this C++ code takes three AST nodes and generates C++ code and an AST based on the input nodes. It iterates over the global variable declarations, checks the data type of the variables being declared, and adds the appropriate code to the AST and C++ streams. It then iterates over each variable in the declaration, adds it to the symboltable map with its data type and array size, and updates the AST and C++ streams accordingly. If the current node represents a function declaration, the function sets the necessary variables, adds the function to the symboltable map, and iterates over the function's arguments to add them to the symboltable map as well. The function then updates the AST and C++ streams with the appropriate code for each argument. Overall, the parse function effectively translates AST nodes into corresponding C++ code and AST representations.

Construction of language

considered ';' after `assign_stmt,read_stmt,write_stmt,cond_stmt,func_stmt` `statements.`

Supports negative numbers, and operators (**+ , - , * , / , == , != , < , AND, OR, NOT**) and expressions can contain numbers,variables,arrays and operators and proper opening and closing parentheses.

Language supports **if-else , while, write, read, function call, assignment** statements**.**

Inside the **main block and any function body ,** return type should be integer, then followed by either of the statements **if-else , while, write, read, function call, assignment.**

write statements must start with opening and closing parentheses.

Language supports **integer, boolean** data types.

It supports function declarations in the global declarations section.

It does support function declarations(including main function) with arguments,local variables followed by statements and then followed by return statements.

It does support recursive function calls and normal function calls.

In global declaration first we need to write the **data type then followed by variable names, array (size mandatory)and function declarations** and separated by commas

---

Limitations

Code should not contain any keyword as variable name,

key words :

main, return, begin, end, read, write, integer, boolean, decl, enddecl, if, then, else, endif, do, while, endwhile, for, AND, OR, NOT ,true, false.