Monsoon 2023 CS1319 A4

Bhumika Mittal

Introduction

The Lexical Grammar and the Phase Structure of nanoC is already defined in the previous assignments. We write a translator for nanoC in C++.

Design Choices

With reference to the details given in the assignment document, we will store the quads in a array and emit the 3-Address Code towards the end.

We augment the grammar with the markers M and N as empty marker and guard to manage the control flow. Each symbol is mapped to either E (expression type) or S (statement) besides a few others which are mapped to symbol pointer, integer value, etc.

We also create a global and current symbol tables to store the variables along with their meta data and the temporary variables which are created during the runtime. Based on the following design choices and changes, we update the grammar and also add the appropriate semantic actions to the Bison file.

Changes in Lexer

I have also changed the working of lexer as follows:

1. **String Literal (`STRING_LITERAL`):**
    - The handling of string literals remains the same. The content of the string literal (`yytext`) is assigned to `yylval.char_value`, and the token type `STRING_LITERAL` is returned.
2. **Identifier (`IDENTIFIER`):**
    - The processing of identifiers also remains unchanged. The `lookup` function of the symbol table (`ST`) is used to find the symbol corresponding to the identifier, and the result is assigned to `yylval.symPtr`. The token type `IDENTIFIER` is then returned.
3. **Integer Constant (`INTEGER_CONSTANT`):**
    - The handling of integer constants appears to remain the same. The content of the integer constant (`yytext`) is converted to an integer using `atoi` and assigned to `yylval.intval`. The token type `INTEGER_CONSTANT` is returned.

4. **Character Constant (`CHARACTER_CONSTANT`):**
    - The processing of character constants also remains unchanged. The content of the character constant (`yytext`) is assigned to `yylval.char_value`, and the token type `CHARACTER_CONSTANT` is returned.

Header file

Let's now describe each class and it's attributes used in the header file:

`sym` Class:

- Represents an entry in the symbol table.
- Members:
    - `name`: Symbol name.
    - `type`: Pointer to the type of the symbol (`symbolType`).
    - `size`: Size of the symbol.
    - `offset`: Offset of the symbol.
    - `nested`: Pointer to a nested symbol table.
    - `val`: Initial value of the symbol.
- Methods:
    - `sym(stringSym, stringSym t = "int", symbolType *ptr = NULL, int width = 0)`
    - `sym *update(symbolType *)`: Updates the symbol table entry.

`symbolType` Class:

- Represents the type of a symbol.
- Members:
    - `type`: Type of the symbol.
    - `width`: Width or size of an array (default is 1).
    - `arrtype`: Pointer to the type of the array elements.
- Methods:
    - `symbolType(stringSym, symbolType *ptr = NULL, int width = 1)`

`symTable` Class:

- Represents a symbol table.
- Members:
    - `name`: Name of the symbol table.
    - `tempCount`: Count of temporary variables.

  - `table`: List of symbols.
  - `parent`: Pointer to the parent symbol table.
- Methods:
  - `symTable(stringSym name = "NULL")`: Constructor.
  - `s *lookup(stringSym)`: Looks up a symbol in the table.
  - `voidSym print()`: Prints the symbol table.
  - `voidSym update()`: Updates the symbol table.

`quad` Class:

- Represents an entry in the quad array.
- Members:
  - `result`: Result of the operation.
  - `op`: Operator.
  - `arg1`: First argument.
  - `arg2`: Second argument.
- Methods:
  - `voidSym print()`: Prints the quad.
  - `voidSym printType1()`: Prints the quad in a specific format.
  - `voidSym printType2()`: Prints the quad in another format.
- Constructors:
  - `quad(stringSym, stringSym, stringSym op = "=", stringSym arg2 = "")`
  - `quad(stringSym, int, stringSym op = "=", stringSym arg2 = "")`

`quadArray` Class:

- Represents an array of quads.
- Members:
  - `Array`: Vector of quads.
- Methods:
  - `voidSym print()`: Prints the quad array.

`basicType` Class:

- Represents basic types.
- Members:
  - `type`: Vector of type names.
  - `size`: Vector of corresponding sizes.
- Methods:
  - `voidSym addType(stringSym, int)`: Adds a type with its size.

Global Variables:

- `ST`: Pointer to the symbol table.
- `globalST`: Pointer to the global symbol table.
- `currentSymbol`: Pointer to the current symbol.
- `quadArr`: Quad array.
- `bType`: Basic type.
- `instrCount`: Count of instructions.

Other Structs:

- `Statement`: Represents a statement with a list of next statements.
- `Array`: Represents an array with type, location, array symbol, and type information.
- `Expression`: Represents an expression with location, type, true/false lists, and next list.

Function Declarations:

- Various utility functions for handling symbols, types, quads, and expressions.

Some helper functions and their usage

`void updateNextInstr()`

- **Input:** None
- **Output:** None
- **Usage:** Increments the global instruction count (`instrCount`) by 1.

`int nextInstr()`

- **Input:** None
- **Output:** Returns the size of the quad array (`quadArr.Array`).
- **Usage:** Retrieves the current size of the quad array, representing the next available instruction index.

`void addSpaces(int n)`

- **Input:** `n` - Number of spaces to add.
- **Output:** None
- **Usage:** Prints `n` spaces to the standard output. Useful for formatting output.

`void changeTable(symTable *nTable)`

- **Input:** `nTable` - Pointer to a new symbol table.
- **Output:** None
- **Usage:** Changes the global symbol table (`ST`) to the specified symbol table (`nTable`).

`int sizeOfType(symbolType *t)`

- **Input:** `t` - Pointer to a symbol type.
- **Output:** Returns the size of the given symbol type.
- **Usage:** Computes and returns the size (in bytes) of the specified symbol type, considering various data types, arrays, pointers, and functions.

`string printType(symbolType *t)`

- **Input:** `t` - Pointer to a symbol type.
- **Output:** Returns a string representing the human-readable form of the given symbol type.
- **Usage:** Generates a string representation of the specified symbol type, including information about base types, arrays, pointers, and functions.

`void quad::printType1()`

- **Input:** None
- **Output:** Prints the quad in Type 1 format (e.g., "result = arg1 op arg2").
- **Usage:** Used to print a quad in a specific format (Type 1).

`void quad::printType2()`

- **Input:** None
- **Output:** Prints the quad in Type 2 format (e.g., "if arg1 op arg2 goto result").
- **Usage:** Used to print a quad in another format (Type 2), typically used for conditional statements.

Symbol Table related functions and their usage

`sym::sym(string name, string t, symbolType *arrtype, int width)`

- **Input:** `name` - Name of the symbol, `t` - Type of the symbol, `arrtype` - Array type, `width` - Width of the symbol.
- **Output:** None
- **Usage:** Constructs a symbol with the provided name, type, array type, and width. Initializes symbol properties such as size, offset, nested table, and value.

`sym *sym::update(symbolType *t)`

- **Input:** `t` - Pointer to a symbol type.
- **Output:** Returns a pointer to the updated symbol.
- **Usage:** Updates the symbol type, recalculates size, and returns the updated symbol.

`symbolType::symbolType(string type, symbolType *arrtype, int width)`

- **Input:** `type` - Type of the symbol, `arrtype` - Array type, `width` - Width of the symbol.
- **Output:** None
- **Usage:** Constructs a symbol type with the provided type, array type, and width.

`symTable::symTable(string name)`

- **Input:** `name` - Name of the symbol table.
- **Output:** None
- **Usage:** Constructs a symbol table with the provided name. Initializes temporary count.

`sym *symTable::lookup(string name)`

- **Input:** `name` - Name of the symbol.
- **Output:** Returns a pointer to the symbol.
- **Usage:** Looks up the symbol in the symbol table. If found, returns the symbol; otherwise, creates a new symbol, adds it to the table, and returns the new symbol.

`void symTable::update()`

- **Input:** None
- **Output:** None
- **Usage:** Updates the symbol table, computing offsets and updating nested tables.

`void symTable::print()`

- **Input:** None
- **Output:** None
- **Usage:** Prints the symbol table with a neat and formatted layout, including details such as name, type, initial value, size, offset, and nested table.

Quad related functions and their usage

`quad::quad(string result, string arg1, string op, string arg2)`

- **Input:** `result` - Result variable, `arg1` - First operand, `op` - Operator, `arg2` - Second operand.
- **Output:** None
- **Usage:** Constructs a quad with the provided result, operands, and operator.

`quad::quad(string result, int arg1, string op, string arg2)`

- **Input:** `result` - Result variable, `arg1` - First operand (integer), `op` - Operator, `arg2` - Second operand.
- **Output:** None
- **Usage:** Constructs a quad with the provided result, integer operand, and operator.

`void quad::print()`

- **Input:** None
- **Output:** None
- **Usage:** Prints the quad based on the operator type, distinguishing between arithmetic, logical, assignment, unary minus, logical NOT, return, goto, label, param, and call operations.

`void basicType::addType(string t, int s)`

- **Input:** `t` - Type to add, `s` - Size of the type.
- **Output:** None
- **Usage:** Adds a new data type to the symbol type symbolTable (ST) with the given type and size.

`void quadArray::print()`

- **Input:** None
- **Output:** None
- **Usage:** Prints the Three Address Code (TAC) represented by the quad array. Displays the quads with line numbers.

`string convertToString(int num)`

- **Input:** `num` - Integer to convert to string.
- **Output:** Returns the string representation of the input integer.
- **Usage:** Converts an integer to its string representation using stringstream.

`void emit(string op, string result, string arg1, string arg2)`

- **Input:** `op` - Operator, `result` - Result variable, `arg1` - First operand, `arg2` - Second operand.
- **Output:** None
- **Usage:** Emits a quad with the provided operator, result, and operands, and adds it to the quad array.

`void emit(string op, string result, int arg1, string arg2)`

- **Input:** `op` - Operator, `result` - Result variable, `arg1` - First operand (integer), `arg2` - Second operand.
- **Output:** None
- **Usage:** Emits a quad with the provided operator, result, integer operand, and second operand, and adds it to the quad array.

Type Conversion, Backpatching and other Important Functions

`sym *convertType(sym *s, string t)`

- **Input:** `s` - Symbol to convert, `t` - Target type.
- **Output:** Returns a pointer to the converted symbol.
- **Usage:** Converts the type of a symbol to the target type, emitting appropriate conversion code if needed.

`bool compareSymbolType(sym *&s1, sym *&s2)`

- **Input:** `s1`, `s2` - Pointers to symbols for comparison.
- **Output:** Returns `true` if the symbol types are equivalent, `false` otherwise.
- **Usage:** Compares the types of two symbols, considering possible type conversions.

`bool compareSymbolType(symbolType *t1, symbolType *t2)`

- **Input:** `t1`, `t2` - Pointers to symbol types for comparison.
- **Output:** Returns `true` if the symbol types are equivalent, `false` otherwise.
- **Usage:** Recursively compares two symbol types, considering array types.

`sym *gentemp(symbolType *t, string init)`

- **Input:** `t` - Pointer to a symbol type, `init` - Initial value (default is an empty string).
- **Output:** Returns a pointer to a temporary symbol.
- **Usage:** Generates a temporary symbol with the provided type and optional initial value. Adds the symbol to the symbol table.

`void backpatch(list<int> l, int address)`

- **Input:** `l` - List of addresses, `address` - Target address.
- **Output:** None
- **Usage:** Modifies the results of quads in the provided list to the target address.

`list<int> makelist(int i)`

- **Input:** `i` - Integer to create a list with.
- **Output:** Returns a list containing the provided integer.
- **Usage:** Creates a list with a single integer.

`list<int> merge(list<int> &l1, list<int> &l2)`

- **Input:** `l1`, `l2` - Lists to merge.
- **Output:** Returns the merged list.
- **Usage:** Merges two lists and returns the result.

`Expression *convertInt2Bool(Expression *e)`

- **Input:** `e` - Expression to convert.
- **Output:** Returns the converted expression.
- **Usage:** Converts an integer expression to a boolean expression, if necessary, and emits code accordingly.

`Expression *convertBool2Int(Expression *e)`

- **Input:** `e` - Expression to convert.
- **Output:** Returns the converted expression.
- **Usage:** Converts a boolean expression to an integer expression, if necessary, and emits code accordingly.

Reference:

Note that, I read through multiple tinyC compiler stuff to understand how their thing is implemented. Hence, some of the code is inspired from there but I have written everything from scratch. I have also referred to the slides and the dragon book for the same. Also, AI tools like ChatGPT and co-pilot have been used for this documentation and for autocompleting the comments in the code (not the implementation)