CS-1319 - Monsoon 2023 - Assignment 4

Team: Saawan Ka Paani

November 30, 2023

1 Introduction

The NanoC 3-Address Code Translator assignment is a comprehensive exploration into the design and implementation of a compiler for the NanoC programming language. The goal of this project is to develop a robust parser using Bison and a lexer with Flex to convert NanoC source code into an abstract syntax tree (`tNode`). The assignment involves the creation and management of a symbol table (`symbol_table`) to handle variables and functions, while a `quadArray` structure facilitates the generation and storage of 3-address code. The project poses challenges such as nested scopes, abstract syntax tree traversal, and precise type checking, requiring a deep understanding of compiler construction principles and practical application of parsing techniques. Successful completion of this assignment demonstrates proficiency in compiler design and provides valuable insights into the intricacies of translating high-level programming languages into executable code.

2 Lexer and Parser

2.1 Flex Specification (A4.l)

The Flex specification (A4.l) defines the lexical grammar of nanoC using regular expressions. It specifies how the input stream is broken down into tokens, where each token represents a meaningful unit in the nanoC language. The Flex file contains patterns for keywords, identifiers, constants, and other language constructs.

```
[language=C]
1. New Identifier Handling:
[language=C]
{ID} {
    // Create a new data_type object
    yylval.datatype = new data_type();

    // Assign the string yytext to the name attribute of the first element in
    yylval.datatype
    (yylval.datatype)[0].name = strdup(yytext);

    // Return the constant IDENTIFIER
    return IDENTIFIER;
}
```

```
Explanation: When an identifier is encountered, a new 'data_type' object is created.
The lexer assigns the identifier to the 'name' attribute of the first element in
'yylval.datatype' and returns the constant 'IDENTIFIER'.

2. Updated Character Constant Handling:
[language=C]
{CHAR_CONSTANT} {
    // Assign the second character of yytext to yylval.charval
    yylval.charval = yytext[1];

    // Return the constant CHAR_CONSTANT
    return CHAR_CONSTANT;
}


Explanation: For character constants, the lexer now extracts the second character of
the constant and assigns it to
'yylval.charval'. The lexer then returns the constant 'CHAR_CONSTANT'.

3. Updated String Literal Handling:
\begin{lstlisting}[language=C]
{STRING_LITERAL} {
    // Duplicate the string yytext and assign it to yylval.strval
    yylval.strval = strdup(yytext);

    // Return the constant STRING_LITERAL
    return STRING_LITERAL;
}


Explanation: String literals are now handled by duplicating the entire string 'yytext'
and assigning it to 'yylval.strval'. The lexer returns the constant 'STRING_LITERAL'.
```

2.2 Bison Specification (A4.y)

The Bison specification defines the syntax and semantic actions for parsing the nanoC source code. This section provides an overview of the key elements and rules in the Bison file.

2.2.1 Token Definitions

The Bison file begins with token definitions using the `%token` directive. Tokens represent terminal symbols recognized by the lexer and are used in the grammar rules to define the syntactic structure of the language.

2.2.2 Symbol Table Operations

The semantic actions in the Bison file include operations related to the symbol table. The symbol table is crucial for storing information about identifiers, their types, and other attributes. The

actions involve looking up identifiers in the symbol table, generating temporary variables, and handling function calls.

2.2.3 Expression Handling

The Bison specification defines rules for various expressions, such as primary expressions, constant values, string literals, and more. These rules involve creating nodes in the abstract syntax tree (AST), generating temporary variables, and emitting quadruples for intermediate code generation.

2.2.4 Function Calls

Rules for handling function calls are specified, including managing parameters, looking up function names in the symbol table, and generating temporary variables for the result of a function call.

2.2.5 Type Checking and Conversions

The specification includes rules for type checking and type conversions in expressions. It ensures that operations are performed on compatible types and emits appropriate quadruples for intermediate code generation.

2.2.6 Control Flow Statements

Rules for control flow statements, such as `if`, `for`, and `return`, are defined. These rules include generating quadruples for branching and looping based on the conditions specified in the source code.

2.2.7 Translation Unit

The `translation_unit` rule specifies how external declarations are parsed, including both variable declarations and function definitions. It serves as the entry point for parsing the entire nanoC source code.

2.2.8 Error Handling

The `yyerror` function is defined to handle syntax errors during parsing. It provides informative error messages, indicating the type of error and the location where it occurred.

2.2.9 Overview

In summary, the Bison specification combines syntactic rules with semantic actions to define the parsing process for nanoC. It incorporates symbol table operations, expression handling, function calls, type checking, and control flow statements to build a comprehensive parser for the language.

2.2.10 Augmentation and Attributes

Explain any augmentation made to the grammar rules and discuss the attributes assigned to grammar symbols.

3 C Code Overview

The C code presented is a part of a compiler implementation for the nanoC language. It includes the necessary header files, structures, and functions for parsing and generating intermediate code. The key components are described in detail below.

3.1 Global Variables

- `FILE* fptr`: A file pointer used for writing output to "three_address_input.txt."
- `char* currentFunction`: A pointer to a character array representing the current function being processed.

3.2 Symbol Table Record Structure

- `symbol_table_record`: A structure representing a record in the symbol table. It includes fields such as size, offset, nested table pointer, name, type, initial value, and array index.

3.3 Symbol Table Operations

- `assign`: A function to assign the values of one symbol table record to another.
- `print_record`: A function to print the details of a symbol table record to an output file.

3.4 Symbol Table Structure

- `symbol_table`: A structure representing the symbol table. It includes a list of symbol table records, the size of the table, and the current size.
- `create_symbol_table`: A function to create a new symbol table with a specified capacity.
- `lookup`: A function to lookup a symbol in the symbol table.
- `insert`: A function to insert a symbol table record into the symbol table.
- `gentemp`, `gentemp2`, `gentemp3`: Functions to generate temporary symbols.
- `print`: A function to print the contents of the symbol table to an output file.

3.5 Intermediate Code Quad Structure

- `fields_quad`: A structure representing a quadruple in the intermediate code. It includes fields for arguments, result, operation, and symbol table record locations.
- `create_fields_quad`: A function to create a new intermediate code quadruple.
- `destroy_fields_quad`: A function to free memory associated with an intermediate code quadruple.
- `assign_fields_quad`: A function to assign the values of one quadruple to another.
- `print_fields_quad`: A function to print the details of a quadruple to an output file.

3.6 Quad Array Operations

- `quadArray`: A class representing an array of intermediate code quadruples.
- `fill_dangling_goto`: A function to fill the target of a "goto" instruction.
- `backpatch`: A function to backpatch a list of line numbers with a target line number.
- `conv2Bool`: A function to convert an expression to boolean if needed.
- `typecheck`: A function to check the type compatibility of two nodes in the parse tree.
- `compute_width`: A function to compute the width of a type in bytes.
- `make_param_list`, `merge_param_list`: Functions to create and merge parameter lists.
- `__construct_quad_list`, `construct_quad_list`: Functions to initialize the quad array.

3.7 Main Function

The `main` function initializes the global symbol table, constructs the quad array, and invokes the parser using `yyparse`. It then prints the symbol table and the quad array to an output file.

4 Header File: parser.h

The `parser.h` header file is a crucial component of the NanoC 3-Address Code Translator project. It declares various data structures, enums, and function prototypes essential for parsing NanoC source code, managing symbol tables, and generating 3-address code.

4.1 Enums

4.1.1 `quad_data_types` Enum

This enumeration represents the different types of quadruple data operations used in the 3-address code. It includes arithmetic operations, logical operations, unary operations, and control flow operations like `goto`, `if-goto`, and more.

4.1.2 `date_types` Enum

The `date_types` enum represents the data types supported in NanoC. It includes primitive data types (`INT_`, `CHAR_`, `DOUBLE_`, `BOOL_`), pointers (`PTR`), arrays (`ARRAY`), and functions (`FUNCTION`).

4.2 Structs

4.2.1 `lnode` Struct

The `lnode` struct represents a linked list node used for managing lists of line numbers. It includes an `index_list` field representing the line number and a `next` field pointing to the next node in the list.

4.2.2 `tNode` Struct

The `tNode` struct represents a tree node used for constructing the abstract syntax tree (AST). It includes `up` and `down` fields representing the upper and lower bounds of a data type, an `l` array for handling arrays, and a `r` field pointing to the right subtree.

4.2.3 `symbol_table_record` Struct

The `symbol_table_record` struct represents a record in the symbol table. It includes fields for the variable or function `name`, `type`, `initial_value`, `size`, `offset`, `array_ind`, and a pointer to a `nested_table` representing a nested symbol table.

4.2.4 `symbol_table` Struct

The `symbol_table` struct represents a symbol table containing an array of `symbol_table_record` instances. It includes fields for the table `size`, current `curr_size`, current `curr_offset`, `name`, and a `return_ind` indicating if the table represents a function.

4.2.5 `parameter_list` Struct

The `parameter_list` struct represents a linked list node used for managing parameters in function signatures. It includes a `parameter` field representing a symbol table record and a `next` field pointing to the next node in the list.

4.2.6 `fields_quad` Struct

The `fields_quad` struct represents a quadruple in the 3-address code. It includes fields for `arg1`, `arg2`, `res`, `op` (operation), and pointers to `arg1_loc`, `arg2_loc`, and `res_loc` representing the corresponding symbol table records.

4.2.7 `quadArray` Struct

The `quadArray` struct represents an array of `fields_quad` instances, forming the 3-address code.

4.2.8 `attribute_expression` Struct

The `attribute_expression` struct represents attributes associated with expressions during parsing. It includes fields for `loc` (location in the symbol table), `TL` (true list), `FL` (false list), `NL` (next list), `type`, `array`, `loc1`, and `val` (attribute value).

4.2.9 `data_type` Struct

The `data_type` struct represents the data type of a variable. It includes fields for `name`, `type`, `pointer_indicator`, `size`, `value`, `loc`, and `array_ind`.

4.2.10 `parameter_type` Struct

The `parameter_type` struct represents the parameters in a function signature. It includes fields for `parameters` and `names`.

4.3 Function Prototypes

The header file declares several function prototypes that are crucial for the parsing and translation process. These include functions for constructing nodes, handling lists, performing type checking, converting types, and managing symbol tables.

4.4 External Declarations

The header file includes external declarations for variables `global_table`, `curr_table`, and `quad_`, indicating that these variables are defined externally.

5 Conclusion

The NanoC 3-Address Code Translator project provided a comprehensive exploration into compiler design and implementation. Through the development of a parser using Bison and a lexer with Flex, we successfully constructed an abstract syntax tree (`tNode`) representing NanoC source code. The accompanying symbol table (`symbol_table`) facilitated variable and function management, while the `quadArray` structure enabled the generation and storage of 3-address code. Challenges included handling nested scopes, abstract syntax tree traversal, and precise type checking. The assignment provided valuable insights into compiler construction, fostering a deeper understanding of parsing techniques and intermediate code generation. Future improvements could involve optimizations, enhanced error handling, and support for additional language features. Overall, the NanoC 3-Address Code Translator assignment significantly contributed to our practical knowledge of software engineering, particularly in translating high-level language constructs into an intermediate representation.