Monsoon 2023 CS1319 A5

Bhumika Mittal

Introduction

The Lexical Grammar and the Phase Structure of nanoC is already defined in the previous assignments. The machine independent translator which isn't perfect but works for many cases has been designed in the previous assignment as well. Now we will be focusing on generating the assembly code for the $x86$(32 bit) for our nanoC language. This target translator is also written in C++.

Test Cases

For the extra credit part, the program supports char type and arrays but since the test cases uses them inside the main function, error might come which is related to function and not char type/arrays.

Test Cases Passed:

- Test 1, Test 2, Test 3
- EC1, EC3, EC4

Test Cases Failed:

- Test 4 - this is mainly because of the error in the grammar which creates problem with functions
- Test 5 - this is also mainly because of the error in the grammar which creates problem with functions

Design Choices

With reference to the details given in the assignment document, we will restrict the language. The design of the translator is discussed in the assignment document.

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 Bison and translator

I have made some changes in both the Bison and translator from the previous assignment in an attempt to debug the errors. Most of these changes are minor changes and hence not including them here.

ASM Generation

Basic overview: It takes an intermediate representation of a program in the form of quads, processes them, and generates x86 assembly code. It uses symbol tables to manage variables and functions. The translation logic is implemented in the generateASM function, which handles various operations and writes the resulting assembly code to an output file. Let's now breakdown the program and see it's functionality in details:

Global Variables

`labelCount`

Description: Count of labels used in the program.

`labelMap`

Description: A map associating label numbers with their count in the program.

`out`

Description: Output file stream for writing data to a file.

`Array`

Description: Vector of quads representing the intermediate code of the program.

`asmfilename`

Description: Name of the assembly file to which the generated code will be written. The default value is set to "1_A5_quads".

`inputfile`

Description: Name of the input file containing the source code of the program. The default value is set to "test".

Activation Record Function

The `ActivationRecord` function is responsible for managing the activation record (stack frame) for a given symbol table `st`. It assigns memory offsets to symbols within the table based on their categories and sizes.

Input

`symTable *st`: A pointer to a symbol table representing the current scope.

Output

It's a void func, no output as such is returned

Function Logic

- Initializes two offset variables, `param` and `local`, to manage parameter and local variable memory allocation.
- Iterates through the symbols in the symbol table (`st->table`).
- For each symbol, assigns a memory offset based on its category and size.
  - If the symbol is a parameter (`param`), assigns the offset and increments `param` accordingly.
  - If the symbol is not a parameter and not named "RETURN," assigns the offset and decrements `local` accordingly.
  - Skips symbols named "RETURN" to exclude them from the offset assignments.

Assembly Code Generation Function

The `generateASM` function processes quads representing intermediate code and translates them into x86 assembly code. It initializes an array of quads, processes various operations, and writes the resulting assembly code to an output file.

In brief, this orchestrates the translation process from intermediate code to x86 assembly by considering various operations, control flow, and function calls. Each operation is translated (my various if-else statements) into the corresponding assembly instructions, and the resulting code is written to an output file which we mentioned above.

Input

None

Output

The function generates x86 assembly code based on the intermediate code represented by quads and prints it. Void func - no output as such is returned.

Standard output stream

This part defines an overload for the `<<` operator, allowing the standard output stream (`ostream`) to print the contents of a vector. The code uses a template to make the function generic for vectors of any type `T`. Let's check each line

- `template <class T>`: Declares a template with a type parameter `T` for a generic function.
- `ostream &operator<<(ostream &os, const vector<T> &v)`: Overloads the `<<` operator for vectors. Takes an output stream (`os`) and a constant reference to a vector (`v`) as parameters.

- `copy(v.begin(), v.end(), ostream_iterator<T>(os, " "))`: Uses the `copy` algorithm to copy elements from the beginning (`v.begin()`) to the end (`v.end()`) of the vector `v` to the output stream `os`. It uses an `ostream_iterator` to control the output format, separating elements with a space.
- `return os;`: Returns the output stream to allow chaining (e.g., `cout << myVector << endl`).

This overload enables using `cout << myVector` to print the elements of a vector, separating them with spaces.

PS - This is like a nice way to do this and hence used - it's a nice to have not need to have.

I/O Implementation

Functions

`int printStr(char *s)`

Input: `char *s`: Pointer to a null-terminated string to be printed.

Output: Returns the number of characters printed.

Functionality: Prints a string of characters specified by the input parameter `s`. The string is terminated by the null character '\0'. If `s` is equal to "\n", a newline character is printed.

`int printInt(int n)`

Input: `int n`: Integer value to be printed.

Output: Returns the number of characters printed.

Functionality: Prints the integer value `n` to the standard output. No newline character is appended.
It uses an internal buffer to convert the integer to a string before printing.

`int readInt(int *eP)`

Input: `int *eP`: Pointer to an integer indicating the error status (ERR = 1, OK = 0)- as mentioned in the header.

Output: Returns the read integer value.

Functionality: Reads a signed integer from the standard input. The error status is updated through the pointer `eP` (ERR = 1 if input is not an integer, OK = 0 otherwise).

It reads characters until it encounters a tab, newline, or space, converting them into an integer. Error checking is performed to handle invalid input.

References

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.

- GCC IO Documentation: https://gcc.gnu.org/onlinedocs/libstdc++/manual/io.html
- I/O Code Reference: https://github.com/bhumikamittal7/cs1217/blob/master/lab2/printf.c (Note: Access may be restricted)
- x86 Instruction Set Reference: https://www.felixcloutier.com/x86/
- x86 Assembly Guide: https://www.cs.virginia.edu/~evans/cs216/guides/x86.html
- x86 Assembly Language Programming: https://www.cs.fsu.edu/~baker/opsys/notes/assembly.html
- Oracle x86 Assembly Language Reference: https://docs.oracle.com/cd/E19253-01/817-5477/eoiyg/index.html