CS1319 - Monsoon 2023 - Assignment 5

Lexical and Syntactical Debuggers

Aryika Mehrotra and Aaryan Nagpal

This is our document explaining our assignment. We have used GCC Version: `11.4.0`, on an Intel Processor.

MakeFile

We first started with creating a MakeFile and making our pipeline ready as we code. Our code in MakeFile is:

```
.SUFFIXES:

CC=gcc
CFLAGS=-Wall -Werror

SRC=26_A5_translator.c
OBJ=$(SRC:.c=.o)
EXECUTABLE=compiler
TEST=./A5_Tests

all: build

compiler: 26_A5.y 26_A5_translator.c 26_A5.l
bison -d 26_A5.y
flex -o lex.yy.c 26_A5.l
gcc 26_A5_translator.c 26_A5.tab.c lex.yy.c -lfl -Werror -o compiler

build: compiler

test: compiler tac asm

open: build
./$(EXECUTABLE) < 26_A5.nc

%.o: %.c
$(CC) $(CFLAGS) -c $< -o $@

tac: build
for i in 1 2 3 4 5 6 7; do \
echo "Running test case $$i"; \
./$(EXECUTABLE) 1 $(TEST)/test$$i.nc > $(TEST)/tac/quads$$i.out; \
```

```
done

asm: build
for i in 1 2 3 4 5 6 7; do \
echo "Running test case $$i"; \
./$(EXECUTABLE) 2 $(TEST)/test$$i.nc > $(TEST)/asm/asm$$i.asm; \
done

prog: build
for i in 1 2 3 4 5 6 7; do \
echo "Running test case $$i"; \
./$(EXECUTABLE) 2 $(TEST)/test$$i.nc > $(TEST)/asm/asm$$i.asm; \
as -o $(TEST)/asm/asm$$i.o $(TEST)/asm/asm$$i.asm; \
gcc -o $(TEST)/prog/prog$$i $(TEST)/asm/asm$$i.o -no-pie; \
done

clean:
rm -f $(OBJ) $(EXECUTABLE) output.asm
rm -f lex.yy.c 26_A5.tab.c 26_A5.tab.h translator
rm -f A5_Quads/* A5_ASM/*
rm -f test*.o test*
rm -f $(TEST)/tac/* $(TEST)/asm/* $(TEST)/prog/*

.PHONY: all build test clean
```

Right now, all are empty files. We used GCC to compile our final code in C.

Parser & Lexer File

Our Parser and Lexer was almost similar to the one we used for A4 with a few bugs that we fixed for this Assignment.

translator.h & translator.c File

Our Symbol, Quad and TAC functions were same as we used for the last assignment.

AR Function

This function is responsible for managing and populating the Activation Record (AR) for the given symbol table. The function takes a single argument- the symbol table.

```
int localOffset = -4;
int paramOffset = 8;
```

We use two integer variables, localOffset and paramOffset, to manage the offsets for local variables and parameters within the activation record.

```
    symbol *temp = table;
    while (temp->next != NULL){
```

```
        if (temp->category == RETVAL){
            temp = temp->next;
            continue;
        }
        else if (temp->category == PARAMETER){
            insertAR(temp->ST_AR, temp->name, paramOffset);
            paramOffset += temp->size;
        }
        else if (temp->category == FUNCTION){
            AR(temp->nested_table);
        }
        else{
            insertAR(temp->ST_AR, temp->name, localOffset);
            localOffset -= temp->size;
        }
        temp = temp->next;
    }
```

We iterate through each entry in the symbol table. This iteration is used to process each symbol (variables and parameters) to determine and assign appropriate memory offsets within the activation record.
Processing Each Symbol: For each symbol in the table, the function checks its category (PARAMETER, FUNCTION, RETVAL) and performs different actions for each:

- Parameter: It calculates and assigns an offset for parameters of a function.
- Function: It calls itself recursively (`AR(temp->nested_table)`) to process the nested symbol table for the function, which manages the activation record for the inner function.
- Others: It calculates and assigns an offset for local variables or other categories of symbols.

Updating Activation Record (AR):
The function updates the activation record with the calculated offsets for each symbol. This involves storing the offset information in a data structure associated with each symbol (`temp->ST_AR`).

Supporting Functions for the AR:

1. `insertAR`: This function inserts an entry into the activation record hash table. It uses `table[]`- an array of pointers to `ARHash`, which is a hash table used for storing activation record entries. It calculates a hash value based on the `key`, looks for the correct slot in the `table[]` array, and inserts a new `ARHash` entry with the provided `key` and `value`. If an entry with the same `key` already exists, it updates the value.
2. `printAR`: The function iterates over the symbols in the given `table` and their associated activation record hash tables. For each symbol, it prints the identifier (key) and its corresponding offset/value from the AR hash table. If a symbol has a nested table (indicating a function scope), the function recursively prints the AR for that nested table.
3. `searchAR`: It calculates the hash value for the `key`, then searches the corresponding hash table slot for the entry with that `key`. If found, it returns the associated value/offset; otherwise, it returns -1, indicating the `key` is not present.

4. `insertLabel`: It calculates a hash value from the key to determine the index in the label table where the entry should be inserted. It traverses the list at that index to find if the key already exists. If it does, it updates its value. If the key is not found, it creates a new `labelHash` node, assigns the key and value, and inserts it into the list at the calculated index.

5. `getLabel`: It calculates the hash from the key and searches the corresponding list. Returns the `labelHash` node if found, otherwise `NULL`.

ASM:

We made the `genASM` function, responsible for generating assembly code from a set of quads, stored in `QuadArray`

1. Labels We identified and processed labels needed for control flow in the assembly code-`insertLabel(labelTable, inst, i)`: Inserts labels into a label table for control flow management in assembly, `getLabel(labelTable, i)`: Retrieves the label from the label table.
It iterates through `QuadArray` and checks for operations that involve labels. For each such operation, it extracts the label number from the result field and inserts it into a label table using `insertLabel`. It then iterates through the label table to assign a unique `labelCount` to each label, which helps in label management during assembly code generation.

2. Global Variable and Data Section
Handles global variables and constants, creating assembly directives for them. `insertGlob(globalVars, sym->name, true)`: Inserts a global variable into a table for tracking.
Iterates through the `global_table` (symbol table for global variables). Depending on the variable type and whether it's initialized, it generates appropriate assembly directives to allocate memory and define global variables.

3. String Literals
If there are string literals (checked using `string_count`), it creates a `.rodata` (read-only data) section in the assembly. Each string literal is assigned a label (like `.LC%d`) and defined with a `.string` directive.

4. Text Section and Function Definitions
Initializes the `.text` section for the assembly code. Creates a parameter list to manage function call parameters. Sets the current symbol table to the global table. Iterates through each quadruple in `QuadArray` to generate corresponding assembly instructions.

5. Assembly Instruction Generation
For each quadruple, depending on its operation (like `PLUS, MINUS, MULT, DIV`, etc.), it generates the appropriate assembly instructions. It handles arithmetic operations, logical operations, control flow (jumps, branches), function calls, and other instructions. Uses register and memory operand addressing as required, handling global variables and stack-based local variables.

6. Function End and Metadata
After processing each quadruple, it generates function end directives and adds size metadata for functions.

Test Cases

All our test cases are running except Test Case 5.
These are our observation of how and why each one of them works/doesn't work.
Test 1: Doesn't work because it does not have a main() function
Test 3: Function being used before the declaration, therefore won't work.
Test 6: The grammar we are using doesn't have a while loop. Therefore this test case won't work.
Test 5: Wrong value of register- we were unable to locate this. Therefore it isn't working for us.
Extra Credit We attempted the Extra Credit portion from 1-6.

References

We started with making the AR but kept getting `segfault`. Then we asked group 4 their design and they said they used Linked Listed hash tables, while we were using hash table array.
As a desperate last attempt we consulted group 4 ONLY for their design of the auxiliary functions, which had used arrays and hash tables. We incorporated this data structure for all our new tables with our architecture.
