Team

Bishoy Nader Fathy Beshara	22
Amr Mohamed Nasr Eldine Elsayed	45
Marc Magdi Kamel Ayoub	52
Michael Raafat Mikhail Zaki	54

Compiler Phase III : Java Byte Code Generator

12th May 2018

MAIN FILES

1. Java.l: represents a lexical analyzer to return the token type to the next phase.
2. Java.y: represents the whole bison code contains the grammar rules along with the semantic action for each one to end up with the full java byte code.
3. Grammar_structs.h: helper structs for carrying needed attributes associated with tokens.
4. label.h/label.cpp: helper files for generation of labels.
5. Types.h: contains Enum defined to handle data types (int, float, boolean, error).
6. Main.cpp: Entry point to the application.

COMMENTS ABOUT TOOLS USED

1. FLEX

It takes the [.l] file and produces a lexical analyzer from it.

2. BISON

It takes the [.y] file and produces a grammar parser from it.

3. JASMIN

We used to generate from our java bytecode output file a .class file that we can run it by java compiler to test our compiler.
After we generate our java bytecode file we added multiple lines to put our code inside a class and a main to be able to be converted by Jasmin.

We use this command (java -jar jasmin.jar prog.j) to convert java bytecode file into .class, then we use this command (java prog) to run the program and test it.

FUNCTIONS EXPLANATION

void back_patch(vector<string *> **list, string m);
Begins patching all strings in list by adding the string m to their end. Useful to add the actual label for goto statements that we don't know their value in their moment of creation in code.
bool need_label(vector<string *> *list);
Checks if this list has instructions or even needs to add a label to them.
void add_to_list(vector<string *> *dest, initializer_list<vector<string *> *> list);
Merges multiple lists and add them to the end of dest sequentially as given to the function.
void clear_scope(uint before_scope_mem);
Clears and removes all variables and return our memory pointer that points to the next empty space for local variables to the location before_scope_mem.
void add_entry(string s, TYPE type);
Add a new local variable with its type.
bool hasId(string s);
Checks if an id is already declared.
TYPE getType(string s);
Returns the type of a specific variable.
uint getMemoryPlace(string s);
Returns the memory address of a specific variable.
void print_code(vector<string *> * code);
Prints the code to a specified output stream.
void perform_label_adding(vector<string *> *code, vector<string *> **next);
Adds a label to the code if the list next needs backpatching, then backpatches next with that label.
void add_label_to_code(vector<string *> *code, label x);
Add new label to the code.
void process_assignment(string id, Basic_nt * par, B_exp_nt * bexp);
Handels the assignment operation with all its cases and error handling.
void process_arith_op(A_exp_nt * par, A_exp_nt * a, char op, A_exp_nt *b);
Handles arithmetic operations with its error handling.

MODELS AND DATA STRUCTURES

We used structs to encapsulate our data and make them more clear and clean.

Basic_nt: has two vectors, code and next. The code it points to and the items that need backpatching to the label after it.

A_exp_nt: has a type field and code.

B_exp_nt: has a type field, code and pointer to next the items that need backpatching to the label after it.

Block_nt: Every part of the code is represented as block, it has two vectors, code and next. The code it points to and the items that need backpatching to the label after it. And before_block_mem as a pointer to help clear the scope variables.

unordered_map<string, pair<uint, TYPE>> symbol_table;
unordered_map<uint, string> memory_table;

Both were used to keep track and even manage the declared variables and clearing the scope if needed from variables declared in it.

ASSUMPTIONS AND JUSTIFICATIONS

- We added boolean data type to help us with the relational operations.
- We used a unit called Block to handle in scope variables, however even in scope you cannot redeclare variables that were declared from an outer scope.
- We support int, float, boolean data types.
- We support while, for, if, if/else statements.
- We support relational operations, arithmetic operations (+, -), multiplication operations (*, /).
- We assume order of operations (not -> and -> or) and (* , / -> + , -)
- We generate operation even when the operands like [x = 3 + 5], we considered this project an intermediate code generation project which means code optimization is not its responsibility, however a possible approach would be to just check for constant values and calculate them internally in rules if its valid.

BONUS

1. **Implementing FOR LOOPS:**
2. **Implementing Boolean Expressions with its operations:**