CSE504 - Compiler Design

Final Project Report: E-- Compiler

By: Arun Olappamanna, Sonam Mandal, Udit Gupta

Introduction:

Compilers are used to translate high level languages to machine code which is understandable by the processor. We have designed and implemented the E-- Compiler, which is a compiler for an event based language. We have implemented various phases of the compiler, starting from Lexical Analysis using FLEX, parsing using BISON, creation of abstract syntax trees, semantic analysis, type checking, intermediate code generation, and final machine code generation.

Features:

Our E-- Compiler has the following features:

- Event handling with parameters
- Recursive function calls
- While loop support
- Short circuit evaluation of logical operators
- Support for shift operators using multiply and divide operations
- Code expansion for unsupported relational operators

Type Checking

We have implemented type checking using two main functions. We have a typeST() function which iterates over all the entries of the Symbol Table and does type checking for global variables, and calls typeCheck() functions for variables, functions, function parameters, local variables, events, rules, patterns, etc. We have a virtual function, typeCheck() which is implemented for all sub-classes in Ast.h and STEClasses.h. The typeCheck() function uses an isSubType() function which we implemented in Type.C, which gives us information about whether a type is a subtype of another type. Using this information, we can do either coersions where required, or throw an error for unaccepted types.

For example:

int a = 3.6;
double b = 5;

In the above, the first statement will throw a typeCheck error since we cannot assign a double to an integer, as double is not a sub-type of integer.
The second statement will result in a type coercion, where 5 will be coerced to be of type double.

Memory Allocation

We have done memory allocation for the following types:

- Global variables: We identify their offset from the DATA section
- Local variables: We identify their offset from the base pointer on the stack (negative offset)
- Parameters: We identify their offset from the base pointer on the stack (positive offset)

We have created a utility function, **MemAllocUtil.C**, which keeps track of the offsets, and resets offsets when we pass it a reset flag (for local variables and parameters). The reset is required for when we enter a new function scope, since the base pointer will change at this point.

We can allocate memory using the following function:

int memAllocUtil(Type *var_type, enum EntryKind entry_kind, int reset_AR);

Intermediate Code Generation

For intermediate code generation, we parsed through the abstract syntax tree and generated intermediate code using the quadruple form of three-address code. Based on the given instruction set, this looked like the best method to use.

The given instruction set consists of the following kinds of operations:

- Arithmetic operators: ADD, SUB, MUL, DIV, MOD, NEG (and their floating point counterparts)
- Bit operators: AND, XOR, OR (no floating point counterparts)
- Relational operators: EQ, NE, GT, GE, UGT, UGE (and their floating point counterparts)
- Print instructions: PRTI, PRTS, PRTF
- Jump instructions
  - Unconditional Jump: JMP <label>
  - Conditional Jump: JMPC <cond> <label> eg. JMPC GE R000 R001 <label>
  - Indirect Jump: JMPI <reg>
  - Conditional Indirect Jump: JMPCI <cond> <reg> eg. JMPCI FGE F001 1.0 R010
- Move instructions
  - Move label to register: MOVL <label> <reg>
  - Move string to register: MOVS <string> <reg>
  - Move integer to register: MOVI <intVal/intReg> <reg>
  - Move float to register: MOVF <floatVal/floatReg> <reg>
  - Move int to float register: MOVIF <intReg> <reg>

    - Move float to int register: MOVFI <floatReg> <reg>
- Load and Store instructions
    - Load int reg from memory: LDI <reg> <val/reg>
    - Load float reg from memory: LDF <reg> <val/reg>
    - Store int reg to memory: STI <reg> <val/reg>
    - Store float reg to memory: STF <reg> <val/reg>
- Input instructions
    - Read a single byte in byte stream: IN <reg>
    - Read integer into specified register: INI <reg>
    - Read float into specified register: INF <reg>

We have created a virtual function **codeGen()** which is implemented for all classes in Ast.h and STEClasses.h files. We have a **codeGenST()** which does code generation at the global scope for global variables and functions. It internally calls **codeGen()** where required.

Register Allocation:

We have created a utility function, **Reg.C**, which is responsible for register allocation. It keeps track of available registers at a global level for both integer registers and floating point registers.

We also allocate special registers for the global data section, for the stack pointer, and for the base pointer. We have a special temp_stack register for doing calculations related to the stack pointer.

The following functions can be used for the above tasks:

int get_vreg_int();
int get_vreg_float();
int get_vreg_global();
int get_vreg_sp();
int get_vreg_bp();
int get_vreg_temp_stack();

Instruction Class:

We have implemented our own Instruction class, **Instruction.[hC]**, which is used to build individual instructions. It keeps track of all the Opcodes, the sources, destination register, labels, conditional statements for conditional jumps, etc.

Example:
The following code snippet is from our Ast.C file, it creates a JMPC instruction, with label, relational operator, sources, etc.

```
immediate0 = new Value(0, Type::TypeTag::INT);
newLabel = Instruction::Label::get_label();
jmpcInstr = new Instruction(Instruction::Mnemonic::JMPC);
jmpcInstr->relational_op(Instruction::Mnemonic::EQ);
jmpcInstr->operand_src1(tempReg, NULL, VREG_INT);
jmpcInstr->operand_src2(-1, immediate0, Instruction::OpType::IMM);
jmpcInstr->label(newLabel);
```

IntermediateCodeGen Class:

The intermediateCodeGen class is responsible for keeping a list of all instructions in order. We have implemented a print function to write the intermediate code to a file. An object of this class is maintained at a global level so that all instructions can be added to it.

Adding an instruction to the list: ***instrList->addInstruction(jmpcInstr);***

Printing the list of instructions: ***instrList->printInstructionList(outFile);***

Implementation of some specific unsupported operators:

Certain operators like relational operators EQ, NE, GE, GT, LT, LE, etc and shift operators SHL and SHR were not supported by the instruction set. Thus to implement these we required more complicated logic. Below is the pseudo-code for how we implemented the above.

SHL:
eg. a << b

```
MOV a R1
MOV b R2
MOV 2 R3
L1: JMPC EQ R2 0 L2
MUL R1 R3 R1
SUB R2 1 R2
JMP L1
L2: /* other code */
```

SHR:
Similar to SHL, except we replace MUL with DIV.

GT:
eg. a < b

```
MOV 1 RDEST
MOV a R1
MOV b R2
JMPC GT R1 R2 L1
MOV 0 RDEST
L1: /* other code */
```

GE, LT, LE, EQ, NE:
These are very similar to GT above.

Implementation of short-circuit code for logical operators:

For logical operators AND and OR we used short-circuit code. Below is the pseudo code for them.

AND:
eg. a && b

```
a->codeGen() /* Store a's result in register: RA*/
MOV 0 RDEST
JMPC EQ RA 0 L1
b->codeGen() /* Store b's result in register: RB*/
JMPC EQ RB 0 L1
MOV 1 RDEST
L1: /*other code*/
```

OR:
eg. a || b

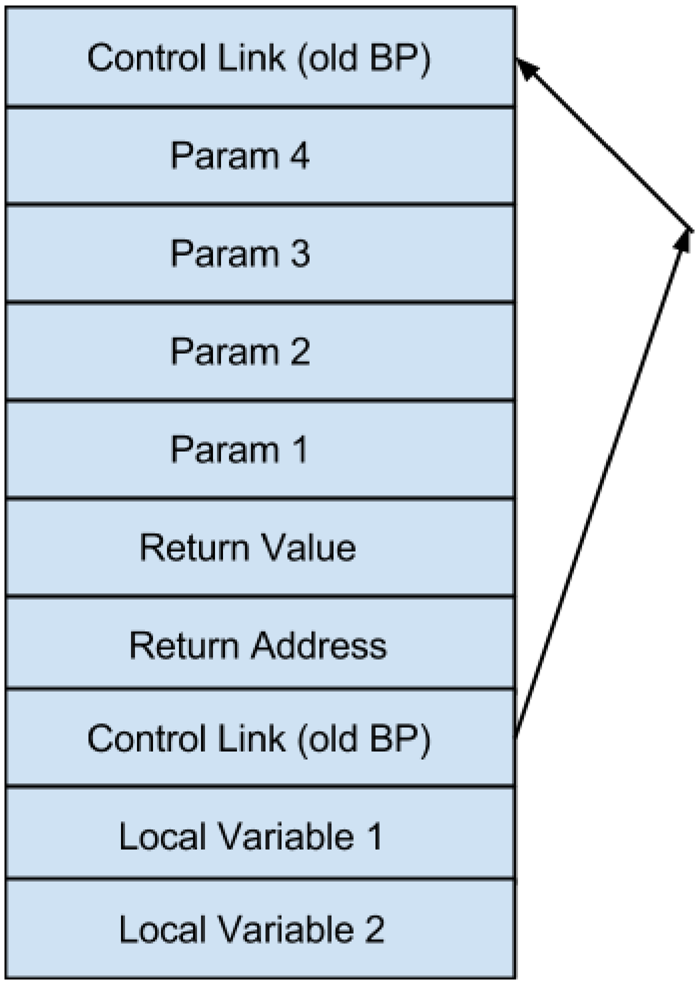
```
a->codeGen() /* Store a's result in register: RA*/
MOV 1 RDEST
JMPC NE RA 0 L1
b->codeGen() /* Store b's result in register: RB*/
JMPC NE RB 0 L1
MOV 0 RDEST
L1: /*other code*/
```

Final Machine Code Generation

We used the icodegen provided to us for converting our intermediate code to machine code.

We have used special registers for the stack pointer, base pointer, global data section (start address), and a special temporary register for doing calculations with respect to the stack.

Our activation record on the stack looks like the following:



At function invocation time, the caller must do the following:

- Push the parameters to the function call onto the stack
- Push the return value
- Push the return address
- Save the old control link (old bp) value on the stack
- Jump to the function

The function, callee, must do the following:

- Save the stack pointer as the new base pointer
- Push it's own local variables onto the stack
- On return, pop the local variables

- Save the return value in the allocated stack position
- Restore the base pointer which is on the stack
- Pop the formal parameters
- Load the return address of the caller.
- Jump back to the caller

Member Responsibilities

Arun Olappamanna:

- Type Check and Type Print functionality for events, pattern nodes, primitive patterns, rules
- Design and implementation of memAlloc()
- Intermediate code generation
- Bug fixing

Sonam Mandal:

- Type Check and Type Print functionality for expressions, and operators.
- Design and implementation of memAlloc()
- Intermediate code generation
- Bug fixing

Udit Gupta:

- Type Check and Type Print functionality for statements
- Implementation of While and Break in Parser and type check
- Design and implementation of memAlloc()
- Writing test cases, testing the functionality.