CSE504 - Compiler Design

E-- Compiler

Final Project Report

Submitted by:

Tejal Kaple	Aniket Alshi	Sahil Parmar	Sohil Gandhi
(109294247)	(109114640)	(109242129)	(109395892)

I. Introduction:

Most of the programming is done in high-level language, while on the other hand the processor understands only machine level language. To bridge this gap we use compilers, which translates high-level language into machine level language. In the process of translation the compiler also checks for errors committed by users during programming.

This project deals with the implementation of compiler design for E--, an event processing language. We have implemented various phases of Compiler Design from Lexical Analysis, Syntax Parsing, Abstract Syntax Tree Generation, Type Checking to Intermediate Code Generation, Code Optimization, and Final Machine Code Generation.

II. Salient Features:

- Multiple Event handling with parameters
- Live variable analysis
- Recursive functions.
- While-break-continue logic supported.
- Code Optimization on High level and IR code.
- Short circuit evaluation of logic operators
- Code expansion of unsupported shift instructions and relational operators
- Compiler options:
    - ***-debug*** to view outputs of all phases.
    - -***noopt*** to compile without optimization

III. Type Checking:

As discussed in the mid report we have implemented type checking using typeCheckST() function which iterates over all the global declarations and definitions. We have implemented a virtual typeCheck() function in all the subclasses of Ast.h and SymTabEntry.h. This function checks for the types of the arguments and gives an error if any incompatible type is encountered. For source operands of different types, coercedType of a particular source operand is updated if one source operand is subType of another source. Coercion is also performed when the actual parameters of the function are subtype of the formal parameters.

Eg:
```
double d1, d2;
int x;
d1 = x*d2;      // Here coerced type of x i.e. coercedType of ExprNode is updated to double
```

IV. Intermediate Code Generation:

Intermediate code generation phase consists of parsing the abstract syntax tree and generating the intermediate code in the form of three-address code. We have used quadruple form for storing the three-address code as it provides us an advantage during the code optimization phase and analysis.

Instructions are basically classified into three categories:

1. Expression Statement: Type of expression statements
   UMINUS, PLUS, MINUS, MULT, DIV, MOD, EQ, NE, GT, LT, GE, LE, AND, OR, NOT, BITNOT, BITAND, BITOR, BITXOR, SHL, SHR, ASSIGN, PRINT.
2. Conditional Jump Instruction
3. Unconditional Jump Instruction
4. Function Call Instructions: enter, leave, param, call

The equivalent three-address structure is stored in intercode class. Symbol table entries have codeGen() function. Labels are managed by a separate labelClass. Code in GlobalEntry is initially converted to equivalent three code and recursively calls for VariableEntry, FunctionEntry, RuleBlockEntry. Furthermore we need to call codeGen for corresponding symbol table entries like refexpression node, ifnode, whilenode, opnode, compound stmtnode, breaknode, valuenode. It will generate a list of intermediate code for complete AST.

Basic Block Creation

In this phase, the intermediate code list is broken down into Basic Blocks. A basic block is a sequential list of instructions without any break in the control flow. We store the list in a Basic Block class, which also stores other information related to basic block.

Information stored in Basic Block is Previous Block labels, Next block labels, Live variables at end of block and live variables at the start of block. This is required for data flow analysis during later phase. At the topmost level we have

Dataflow container class to store list of basic blocks for each function and global block. Within each container we have list of basic blocks and within each basic block we have list of intermediate three-address code.

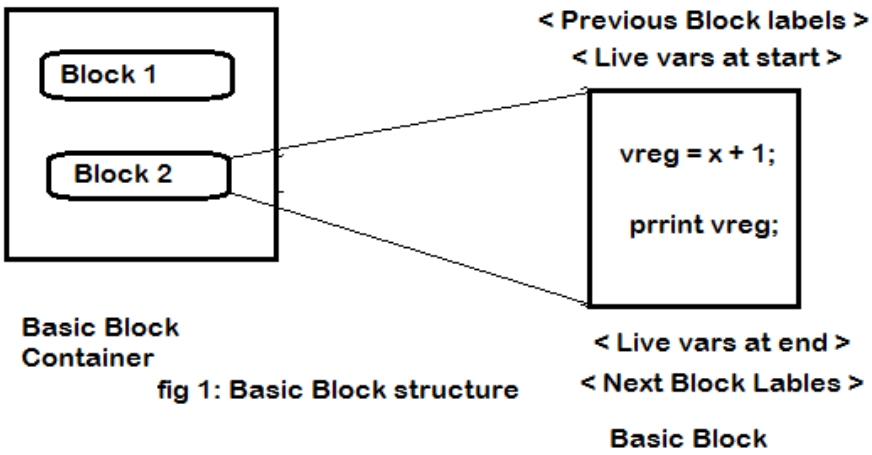


fig 1: Basic Block structure

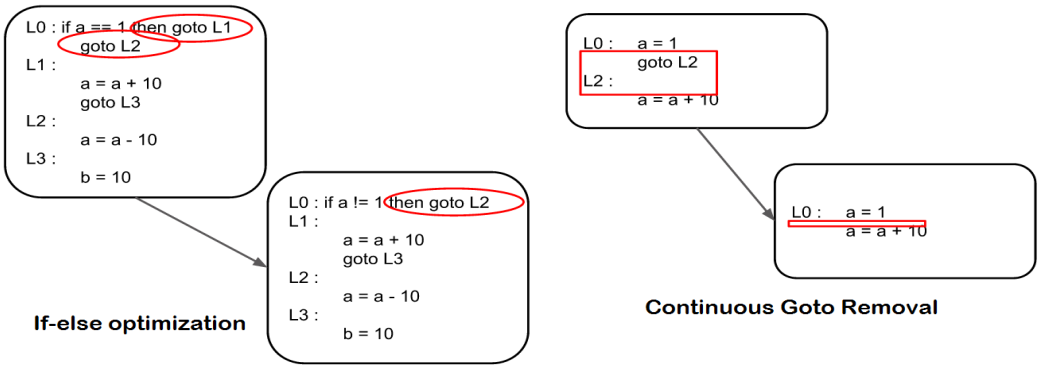
IV. Optimization:

We have implemented following optimizations in our program.

At the level of three-address code we perform following optimizations:
1. If – Else Optimization
2. Continuous Goto – label Removal
3. Unreachable code removal

1. **If-Else Optimization:** In this optimization, we make an attempt to reduce the jumps in the code by avoiding the extra Goto's in the code. We reverse the polarization of the condition in IfNode and introduce only single jump if that condition is satisfied else flow will go the next label present.

2. **Continuous Goto removal:** If in the three address code format, we have Goto and label appearing consecutively then we skip the jump by combining the jump and label. Following code depicts the optimization :



3. **Unusable code removal:** Code after the return statement in the block and code after the Goto inside one basic block is the code, which is never going to get executed because the code is no longer reachable. Once we encounter the goto inside a basic block or return then we ignore all the statements until we encounter the new label or leave statement in case of return.

At level of basic block, we have implemented following optimization:

1. Constant Propagation
2. Constant Folding
3. Zero constant removal
4. Unused function blocks removal
5. Dead code removal using live variable analysis

Constant Propagation:

If we know that value of a variable at compile time and it is not getting altered in any path between its definition and usage, then we propagate the value of variable and replace the occurrence of node directly with corresponding value.

When we encounter a definition of an expression, we insert it into a map of variable and corresponding expression node. If it is getting redefined, we invalidate entry in map and insert new value. Further down the code when we have usage of the variable, we look up its value in map and replace it. We construct new inter code with corresponding values and insert into the block.

Constant Folding

Statically evaluating the value of an expression is called constant folding. In code, we first check if the operands of given three address code are both value Node. Based on the type of the operator of the expression, we extract value of two value node operands, apply operator and evaluate the value. We replace the node with assignment node with value, which we evaluated.
Apart from this, if available we also statically evaluate the condition of the if-Node. If the values of the operands used in the condition of ifnode are available at compile time. If cond is evaluates to true then we directly replaced directly with Goto Label or if else label is present, we introduce Goto to false label.

Zero constant removal

In the three address code, if the value of the operand is zero then we replace the given expression node after evaluating the value statically. This saves us the unwanted push and load in the registers while generating the code.

Unused function blocks removal

If the basic block container function is not reachable from anywhere in the program i.e. the function block is not getting called then we remove the function and all corresponding three address codes within the basic block container of that function.

Dead code elimination

We perform the control flow analysis and identify the live variables at the start and end of each basic block. Based on this information, we identify variable definitions, which are not live and remove the corresponding three-address code within the block.

V. Data flow analysis:

We have extensively performed data flow analysis on basic blocks structures to populate the live variables at the start and end of each block. We can leverage this information further for dead code elimination. Our approach of data flow analysis can be summarized as followed:

Keep a bit vector to check which blocks are visited. For each block we already have list of previous and next block labels available.

1. Start from the block, which has next label list as null. For this block, traverse three address code instructions from bottom to top. For every usage of a variable, push it in used vars list and set bit as 1. Going up if we encounter a variable definition, we check in bitmap. If var is present set the bit as zero. If not, we check in live variables at the end of block and check for this variable, if yes we unmark that variable. At the top of the block, all the variables, which are in used vars list, which are still unmarked, and vars from end-live vars, which are unmarked, are still live at the top of basic block.

2. Propagate the values of live vars at top of a base block to the set of live variables alive at bottom of its parent. Iteratively call all its parents. Continue till you either reaches a block, which has, previous label null or there is recursion and values of live variable list stops changing. i.e. values have converged.

3. Check if any basic block is still unvisited, repeat from step 2.

Once we have list of live variables for each block, dead code elimination can be performed. For each block, when a definition of a variable is encountered, check for that variable in set of used vars in that block. If present invalidate the vars. Else check in live vars at the bottom of block else check if variable is global. If neither of three conditions is satisfied, then remove the variable from the block.

VI. Event Handling:

Following is the flow of event execution:

- ✓ eventHandler() method of GlobalEntry class iterates over the vector of RuleNode
- ✓ After iterating over the RuleNode vector a state transition as below is obtained
- ✓ On receiving user input, it is compared to the set of existing equivalent ascii name and a jump is performed to the corresponding event body.

```
_S_END: IN R005
        //Dummy Instruction
  _S_START: PRTS "\nEnter Event Name ('0' for
      exit): "
  IN R005
  JMPC EQ R005 97 _S_a
  JMPC EQ R005 48 _S_EXIT
  PRTS "Invalid Event Name\n"
  JMP _S_END
```

```
JMP begin
_S_a: MOVL _S_END R005
STI R005 R000     // Return Addr Pushed on Stack
SUB R000 4 R000
PRTS"Enter Param 1: "
INI R005        //READ Event Parameter Input
STI R005 R000       // Parameter Pushed on Stack
SUB R000 4 R000
PRTS"Enter Param 2: "
INI R005        // READ Event Parameter Input
STI R005 R000    // Parameter Pushed on Stack
SUB R000 4 R000
JMP _event_a
```

VII. Final Machine Code Generation:

Input to final CodeGen:

To the function finalCodeGen() we have an input of BasicBlockContatiner object. This class contains a vector of Basic Block objects. This InterCode class deals with quadruple format of instructions. We iterate on the vector of Basic Block objects. Each Basic Block contains a vector of InterCode class objects.

Special Registers:

We have pre-allocated few special registers like:

Stack Pointer	<RSP>	R000
Return Value	<RRV_I> <RRV_F>	R001 F001
Shift Operations	<RSH_Cnt> <RSH_Val>	R002 //for count R003 //for value
Address	<RRET_ADD>	R004 //stores return address of the caller function
Event Params	<R_PARAM> <F_PARAM>	R005 //user input taken in these two registers F005

Functions:

When function calls are made the passing of control from caller function to callee function takes place as follows:

Caller's Responsibility (On Function Invocation)

- Push local and temporary variables
- Push return address
- Push actual parameters
- JMP to callee

Callee's Responsibility (function definition)

- Pop formal parameters
- Load return address of caller
- JMP to caller

On stack it can be visualized as:	Equivalent ASM:
Push local and temporary variables.	SUB RSP 4 RSP STI/STF reg_name RSP
Push return address	SUB RSP 4 RSP MOVL callerLabel reg_name STI reg_name RSP
Push actual parameters	SUB RSP 4 RSP STI/STF reg_name RSP
Jump to callee label	JMP callee__
Pop formal parameters	ADD RSP 4 RSP LDI/LDF RSP reg_name
Load return address	ADD RSP 4 RSP LDI RSP RRET_ADDR

VIII. Future Work:

a. Enhanced Event Handling by supporting PAT nodes with sequence operators.
b. Optimization like Common Sub expression Elimination, Code Propagation, Function Inlining and Loop Invariant Code Motion etc can be implemented.
c. Optimized Register Allocation Policy in Final Machine Code Generation
d. 3 address code generation for class structures.

IX. Member Responsibilities:

Sahil Parmar	**Tejal Kaple**
1. Event Handling 2. Intermediate code Generation 3. Final Code Generation	1. Type Checking 2. Final Code Generation 3. Register Allocation
Aniket Alshi	**Sohil Gandhi**
1. Type Checking 2. Intercode Generation 3. Constant Optimizations 4. Live Variable Analysis	1. Intercode Optimizations 2. Constant Optimizations 3. Basic Block Optimizations