Stony Brook University

# Compiler for E–

**Ankur Mittal**
ankur.mittal@stonybrook.edu

**Kavita Agarwal**
kavita.agarwal@stonybrook.edu

**Prashant Pandey**
prashant.pandey@stonybrook.edu

May 23, 2014

## 1 Introduction

In addition to the lexical and semantic analysis, the compiler has been extended to include Type Checking, Intermediate Code generation, Optimizations and Machine code generation.

## 2 Design

We followed a modularized approach so the given code structure was extended further so as to implement intermediate code generation. Following new classes were added:-

### 2.1 Intermediate Code Generation:-

- **RegMgr** – A class that manages the assignment and purging of registers. It facilitates book-keeping to keep track of what registers have been used and which have to be spilled over. It creates a map between variable entry and register names.
- **Quadruple** – A class which contains 3 address code quadruple object corresponding to the intermediate code generation.. A set of quadruples correspond to the intermediate code of the program. The quadruples involve temporary variables. Maximum no of optimizations have been applied on the quadruples objects.
- **Instruction** – A class whose objects have assembly code corresponding to the intermediate code generation. The instruction set contains machine level instruction, a maximum of three parameters , label and comments (if applicable)

### 2.2 Machine Level Code Generation :-

- **Program Code** – A class which consists of Code Modules
- **Code Module** – A class which consists of set of instructions from which the corresponding machine level code is generated.

The sub-classes of AST have been extended to return the object of Quadruples whenever Intermediate code gen is called on them. A list of Quadruples are returned to Function Entry and RuleNode classes. Optimizations are performed on these quadruples to finally generate low level machine code.

### 2.3 Memory Layout:-

- Global variables, locals and temporaries are all stored registers. (This feature could be enhanced to implement a particular threshold number of variables to go in registers and rest on stack so as to ensure optimal register utilization)
- Registers are pushed on stack during flushing operation.
- Special Purpose Registers
  - R000 - Stack Pointer
  - R001 - Base Pointer
  - F000 – contains float return value register
  - R002 – contains int return value register

## 3 Compiler Specifications

A compiler for event based language E– has been implemented which includes following functionality.

1. **Type Checking Support :-** Supports type checking for events, functions, expressions, if-else, while-break constructs as explained below.
2. **Event Invocation :-** Supports events by matching first character only. It supports 53 events [A-Za-z] and an "any" event. It supports multiple events and their invocations. The input to the events are passed through and input file which contains hexadecimal equivalent of input in a character format.
3. **Function Invocation :-** Supports parameterized and non-parameterized function calls. A function is called either from an event or from a function. The return value of the function is saved in R002 or F000 depending on whether the return value is "int" or "double". If the return type is void then both these registers contain a default value of zero.
4. **Flushing of registers :-** This compiler for E—also supports flushing of register values. As functions are statically generated and they occupy some registers we push their state on stack so that their state can be restored whenever needed and these registers can be used for other evaluations as well.
5. **Recursion :-** Function calls have been implemented in a way which supports recursive calls as well. The return value of the function call is either present in R002 or F000 as explained above.
6. **Short circuit operators for expressions :-** Supports short circuit operators on expressions so that number of instructions executed are less in case of logica AND and OR operators. Also integrated these with "if" and "while" constructs.
7. **If-then-else :-** Supports If then else constructs along with short circuit operators. This helps in conditional branching and helps facilitating recursion (base case execution).
8. **Nested while support with break :-** Supports multilevel and same level while constructs. It also checks for the degree and level of nested ness for while which extends to support break for coming out of nth level of while. (n – no of nested levels supported for while)
9. **Implementation of SHL, SHR :-** Supports shift left and shift right operators by using multiplication and division respectively.
10. **Optimizations :-** All the optimizations were performed at **intra block level**.

- Common sub expression elimination :- The expressions which are common across quadruples have been removed. This reduces the number of temporaries used.
- Removing redundant assignments to same temporary variable which further reduces the no of temporary registers used.

11. **Print Support :-** Supports print functionality by adding grammar rules. Type checking for "prt" have also been added. Thus print in E—works with following syntax "prt-¿" on similar lines with cout of C++.

# 4 Type Checking

It is done in a separate pass over the AST. Following are different kinds of checking which we plan to implement in our code:

- Type check on "FUNCTION/EVENT" declaration and invocation
  - Mismatch in number of arguments
  - Mismatch in order of arguments
  - Mismatch in type of arguments (consider sub-types)
  - Function return type mismatch (conside sub-types and void)
- Expression evaluation type check
  - Variable initialization type check
  - Type mismatch between LHS and RHS of expression (consider sub-types)
  - Implicit type conversion (coercion) during expression evaluation
- Type check for conditional expression ("IF/WHILE")
  - Type of conditional expression should be boolean
- Operator type checking in expressions
  - Type mismatch in the operands for an operator
  - Assignment sub-type error. (e.g. Assigning a double value to an integer value)
  - Number of operands for a given operator
- Event pattern type checking
  - Check if an event pattern is negatable or not. Event patterns containing sequence operator i.e ":" and "**" are not negatable
  - Verify the number of event pattern operands for a given operator
  - Event parameters are const by default, so they should not be modified in the rule body.

# 5 Roles and Responsibilities

The development of the compiler was divided into multiple phases. And the roles and responsibilities of each member spaned across phases. We split the work among the members during each phase of development. The various phases of building this compiler are divided into five major categories:-

- Project Design :- In this phase we did brain-storming discussions to decide upon the classes and their structures for implementing all the below mentioned features. Deliverable of this phase was a class diagram for all compiler functionalities.

- Type checking :- This phase had various different tasks like:-
  - EntryLevel type check: - Type checking on subclasses of SymTabEntry like EventEntry, FunctionEntry, RuleblockEntry, WhileBlockEntry, VariableEntry which was done by Ankur.
  - Statemnet Nodes Type check:- It included type checking for WhileNode, BreakNode, ReturnStmt, BreakStmt, IfNodeStmt which was done by Prashant.
  - Expression Node Type check:- It includes type checking on RefExprnode, ValueNode, OpNode, Invocation Node which was done by Kavita.
- Intermediate Code Generation :-
  - Register Manager : Managing book keeping, purging registers and mapping of registers with corresponding variable entries. This was done by Ankur
  - Code Module, Program Module, IntercodeElem, InterCodeParam, Quadruples, Instruction:- These classes and their structures was laid out by Prashant.
  - Generating quadruples for expressions nodes- This was done by Kavita.
  - Statement Nodes - This was done by Prashant.
  - Code Generation on block level GlobalEntry, FunctionEntry and RuleNode - This was done by Ankur.
  - Short circuit Evaluation :- This was done during the Intermediate code generation of OpNode by Prashant.
- Optimization
  - Common Subexpression Elimination :- It was done by Ankur.
  - Reducing Redundant Temperories :- It was done by Kavita.
  - Dead Code Elimination :- It was done by Prashant.
- Machine Code Generation
  - Generating instruction set for function calls :- This was done by Ankur.
  - Mapping of Quadruple opcodes with the actual instruction set- This was done by Prashant.
  - Forming instructions from quadruple set- This was done by Kavita.

# 6 Challenges

- Design of the Compiler Design aspects had to be dealt carefully so as to ensure that use of quadruples and instructions across various classes had a common semantics.

# 7 Future Work

- Event Regular Expression matching to support complex event patterns
- Reaching definitions and live Variable anaylsis so as to eliminate the dead code
- Implementing further optimizations like strength reduction, induction variable, dead code elimination, etc.
- Spill Over of registers