Compiler for E--

Intermediate Project Report

Kavita Agarwal kaagarwal@cs.stonybrook.edu
Ankur Mittal anmittal@cs.stonybrook.edu
Prashant Pandey ppandey@cs.stonybrook.edu

OVERVIEW

In this course project we will work on first 5 phases of compilation. The project aims at building a compiler for E– which is an event based language. We have completed the first two phases during the assignment 1 to 3. In the main project we will use the Abstract Syntax Tree (AST) that we generated during the assignment 3 for further phases of the compilation. All three team members will have role to play in every stage. Following are three major parts of the project:

1. **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.
2. **Memory Allocation** : In this phase we will allocate required memory for the various constructs of the language. The memory allocation is divided into three phases:
   - Static/Global memory allocation
     - Un-initialized data (bss section)
     - Initialized data [writable] (data section)
     - Initialized data [const] (const section) (Not supported in E--)
   - Dynamic memory allocation
     - Stack allocation
     - Heap allocation (not supported in E--)
   - Temporary allocation
     - Assigning a set of variables to registers
     - More than one variable can be assigned to one register
     - But accessing the variables not present in register will require RAM access. Thus compiler optimization through liveness analysis can be performed. Please refer to the set of compiler optimizations for more details.

   As we have no new keyword we would not be handling heap allocation. All allocations on stack can be referenced using offsets from the frame pointer. These offsets can be computed from the symbol table. The different aspects taken into consideration while computing offsets for local variable are whether the local variable is in the stack frame or in lexically enclosing stack frame. But since we are not handling the case of nested procedures therefore we need not deal with lexically enclosing stack frame.
3. **Intermediate Code Generation** :- The aim here is to convert the AST to generate abstract (machine independent) code which is closer to a target language. Certain key points which should be kept in mind while deciding the semantics of code generation are :-
   - **Register assignment** :- Computations using registers are faster but at the same time for large expressions we have to be careful that the number of temporary variables does not exceed the register count.
   - **Intermediate Code Representation** : We will use 3 address code (quadruples) and will follow postfix notation for expression evaluation. For example:-

     Expression:

     $$a := b * -c + b * -c$$

     Intermediate Code:

     $$t1 := -c$$
     $$t2 := b * t1$$
     $$t3 := -c$$
     $$t4 := b * t3$$

Table 1

#	Op	Arg1	Arg2	Res
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	=	t5		a

Implementation of Three-Address Statements: Quads

$$t5 := t2 + t4$$
$$a := t5$$

- **Temporary variables**:- Break complex expressions into autonomous subexpressions, and store their results in a named object. If multiple intermediary results have to be stored sequentially we will reuse the same object. For example:-
  given expression:
  $$int\ s = s1 + s2 + s3;$$
  evaluated as:
  $$int\ temp\ =\ s1 + s2;$$
  $$temp = temp + s3;$$
  $$int\ s = temp;$$

- **Handling If-then-else/Switch/While constructs** :- We will be supporting short circuit operators in conditional constructs like if then else/ switch and while statements. These can be implemented by using labels and jump statements.
- **Function Implementation**:- During function invocations we will support Call By value parameter passing mode. The activation records formed during the function invocation need to follow a set of predefined rules. Thus we define a protocol which would be followed by the caller and callee while pushing contents on to the stack. The layout for the protocol is as follows:-
  - **Caller**
    - Push the argument values for the callee function (from right to left)
    - Push the return address
  - **Callee**
    - Push the current base pointer (bp register). This acts as the control link
    - Copy the stack pointer (sp register) into the base pointer (bp register)
    - Push the local variables onto the stack

RUNTIME ENVIRONMENT

The memory layout consists of a stack which contains activation record for various function invocations. The given base architecture consists of 1000 registers out of which we predefine 4 special purpose registers as follows:-

- Stack Pointer (sp)
- Base Pointer (bp or frame pointer)
- Global Section Base Address- This register contains the base address of the memory from where global variables start. Other global variables can be accessed using respective offsets from this address.
- Return Value- The callee uses this register to store the return value which can be accessed by the caller.

EVENT PATTERN MATCHING

We will use direct DFA construction techniques for event matching. As far as event parameters are concerned we still have certain doubts regarding the same. We have asked them on the discussion forum and are looking forward for an answer. We will try to implement the translation algorithm depending upon understanding and time constraints.

COMPILER OPTIMIZATIONS

1. **Temporary allocation** :-
   Through liveness analysis[1], compilers can determine which sets of variables are live at the same time, as well as variables which are involved in move instructions. Using this information, the compiler can construct a graph such that every vertex represents a unique variable in the program. Interference edges connect pairs of vertices which are live at the same time, and preference edges connect pairs of vertices which are involved in move instructions. Register allocation can then be reduced to the problem of K-coloring the resulting graph, where K is the number of registers available on the target architecture. No two vertices sharing an interference edge may be assigned the same color, and vertices sharing a preference edge should be assigned the same color if possible. Some of the vertices may be precolored to begin with, representing variables which must be kept in certain registers due to calling conventions or communication between modules. As graph coloring in general is NP-complete, so is register allocation. However, good algorithms exist which balance performance with quality of compiled code. The graph coloring technique is so effective because it takes into account not only a variable being considered for register allocation, but also all the variables which are live at the same time. The logic is that if all the neighboring live variables of variable V can be assigned registers, then so can V plus all the neighbors. So it is a recursive case of removing a variable from the set of live variables at a point, called the graph, and then examining the resulting "graph" minus one variable. The loop continues until the reduced graph can be allocated, and all the other variables are spilled to memory.
2. **Return Value** :- We will not store the return value on the stack while function calls. We will follow a convention, that the callee will store the return value in some predefined register before returning the control. For eg under normal conventions the return value is saved in rax register. The called function will read the value from that register and

[1]Muth, Robert. "Register liveness analysis of executable code." Manuscript, Dept. of Computer Science, The University of Arizona, Dec (1998).

assign it as per the code. This way we can avoid unnecessary memory access while reading and writing return value on the stack.

3. **Address Optimization** :- Referencing the global variable by constant address requires two instructions while accessing the same variable through pointer offset requires only one instruction. The number of global scalar variables in many programs is relatively small, and often can fit in a global variable pool. For such programs, all of the global scalar variables (and small global arrays) can be accessed via one pointer and an offset, thus avoiding more expensive load and store sequences and reducing code size. We can store the value of the global pool in one specified register that we can use for faster access.

4. **Dead Code Elimination** :- The compiler will identify the dead code that will not affect the overall execution of the program. This set of code will not contribute to the assembly code generated by the compiler and thereby make the execution of the program faster. For example: variable declared but not used, variable getting assigned multiple values before being used.

5. **Static Optimization** :- If a counter variable (i.e a variable, that is used as a counter in loop) is declared static then during each iteration of the loop, the value of the counter variable will be read through a address lookup. Otherwise if we can change the storage class of the counter variable from static to register then we can execute the loop very efficiently.

6. **Function Inlining** :- The overhead associated with calling and returning from a function can be eliminated by expanding the body of the function inline and further optimization may be performed as well.

```
int add (int x, int y)
{
  return x + y;
}

int sub (int x, int y)
{
  return add (x, -y);
}
```

Expanding add() at the call site in sub() yields:

```
int sub (int x, int y)
{
  return x + -y;
}
```

ROADMAP

1. *Type checking:* This feature has been fully implemented
   - Type check on FUNCTION/EVENT
   - Type check on expression evaluation
   - Type check on expression of IF/WHILE
   - Type check on operators in expressions
   - Type checking on event patterns
2. *Analyzing ICode and the assembly code that needs to be generated:* The code was fully analyzed to understand the intricacies of final assembly code
   - Analzing ICode gave us an idea about the type of assembly that **erun** expects and the instructions allowed in the assembly to be generated
3. *Intermediate Code Generation:*
   - The AST generated in assignment 3 will be used for intermediate code generation using *Three-Address Code* approach
4. *Designing algorithm and strategies for optimizations:*
   - The algorithms for the optimization techniques discussed in above section will be designed in this phase
5. *Code Generation(with memory allocation):*
   - Algorithms and intermediate code generated in previous step will be used for final code generation which in turn will be used as an input to **erun** to generate the executable file
6. *Testing phase:*
   - We will perform unit testing after every phase of the roadmap
   - Integrate all phases and perform integration testing to check the compiler for larger and complex inputs