CSC488 A5

group_0044

December 4, 2019

Our implementation of code generation required a couple new classes (U for updated, A for added):

- U **ArrayDeclPart.java**
  Fixed size computation.
- U **RoutineDecl.java, ConditionalExpn.java, IfStmt.java, WhileDoStmt.java**
  Added intermediate `visit` action for extra codegen actions. Fixed size computation.
- A **PrintExpn.java**
  For ease of distinguishing when we are printing out expressions (i.e. an expression explicitly meant to be outputted).
- A **CodeGen.java**
  An AST visitor. Added checks for each codegen action described in `codegen.pdf`. Then override each visit action on a node type with the correct sequence of calls to the codegen actions (note that some actions are empty/unused). Each call generates intermediate instructions (defined in `IR.java`) to later be translated into machine instructions. As `CodeGen` traverses the AST, it builds a register-based IR while keeping track of various metadata like the lexical level, routine locations, order number of next available register in the current frame, and maintaining a symbol map for the current scope.
- A **IR.java**
  Utility class for meta instructions that can carry with them 1, 2, 3 or 4 operands. Some meta instructions include `INIT_FUNC_FRAME` (i.e. to translate to instructions for the setup of a function).
- A **Translator.java**
  Translates all meta instructions to their respective machine instructions. Over the course of building the `IR`, various metadata was collected to aid in "patching" instructions that needed information from some later part of the code i.e. the address of where to branch to. All of this metadata is made available here and so the code is written and patched accordingly via direct access to the `Machine`.
- A **SymbolMap.java**
  Similar to `SymbolTable` from A3, except slimmed down – it keeps track of variables, their types and their addresses in a scope, and which scopes are children of who. As we traverse

the AST, we switch the `SymbolMap` accordingly to gain access to the appropriate set of symbols.

Significant Changes from A4

- **Register Based IR**
  In preparation for A6, we opted have `CodeGen` build a register-based IR. This means we had a dedicated register for the result of every expression. Every major scope allocates the number of registers is requires at the start, and then it simply uses `LOAD` and `STORE` calls to write to different register offsets and simulate a register based machine. The code generator tracks how many registers are used in each major scope, and inserts code to allocate space for each register.

- **Minor Scope Space Allocation**
  For ease of implementation, storage required for minor scopes are now allocated by its corresponding major scope, which also takes care of its deallocation – previously in A4, the plan was to allocate and pop storage as we encountered minor scopes.

Contribution

- Jasmeet Sidhu
  Focused on designing and implementation of IR and Translator, implementation of codegen rules.

- Jasmeet Brar
  Contributed to IR and Translator implementation and design as well as implementation of codegen rules.

- Kareem Golaub
  Design document. Also contributed to IR and Translator design.

- KyoKeun Park
  Contributed to IR implementation, as well as creation of test cases and testing script.