Project 3: An Intermediate Language and Its Code Generator

Recall that the combined project for the semester is to implement a compiler from Tiger to MIPS, as a sequence of phases:

1. A scanner.
2. A parser.
3. Semantic analyses.
4. A generator from source syntax to intermediate code.
5. A generator from intermediate code to MIPS assembly.

The third project is to implement phase four: to define an intermediate language and generate code from the AST of a well-typed program. For this project, you will extend your compiler so that after parsing and type-checking a given Tiger program, it generates a representation of the program in intermediate code. The project work will consist of three components:

1. A definition of a target intermediate language IL.
2. A compiler phase that takes the AST of a well-typed Tiger program and generates an equivalent program in IL.
3. An interpreter that takes an IL-program P and simulates the effect of P, reading the standard input of the compiler.

The language definition and code generator are essential for the functionality of the full compiler. The interpreter will be used primarily to debug and evaluate your design.

1 Phase Design

1.1 An Intermediate Language

The first component of the assigned compiler phase is an intermediate language. Technically, you can design any intermediate language IL, so long as you implement an `interpret` operation that takes a program in IL and correctly simulates the effect of IL on the standard input of the program. In particular, a language of abstract syntax trees for Tiger could feasibly serve as an intermediate language. However, the final course project will be to implement a compiler phase that translates a program in your intermediate language to an assembly program for the MIPS architecture. Thus, we *strongly* recommended that you design and target an intermediate language in which each program is a linear sequence of low-level instructions.

We discussed one such intermediate language IL in lecture. An IL program is defined over a fixed finite set of arithmetic operators $\texttt{ops}_A = \{\texttt{add}, \texttt{sub}, \texttt{mult}, \texttt{div}, \texttt{and}, \texttt{or}\}$, comparitors $\texttt{ops}_C = \{\texttt{breq}, \texttt{brneq}, \texttt{brlt}, \texttt{brgt}, \texttt{brgeq}, \texttt{brleq}\}$, and infinite spaces of virtual registers `regs` and control labels `labels`. An IL program is a linear sequence of instructions. Each instruction is a combination of virtual registers, labels, and opcodes. The spaces of virtual registers and labels are implemented in the provided classes `Regs` and `Labels`. Both classes have straightforward implementations, but fixing their definitions allows us to simplify the definition of the `State` class, which you can use to implement a significant component of your intermediate language's interpreter (§1.3).

In particular, for all virtual registers `r0` and `r1`, the operation `assign r0,r1` copies the value in `r1` to `r0`.

For all registers `r0` and `r1` the instruction `load r0, r1` loads the value at the address in `r0` into `r1`. The instruction `store r0, r1` stores the value in `r1` into the address stored in `r0`. The instruction `assign r0,r1,r2` allocates an array with size stored in `r0` in which each entry has the value stored in `r1`, and stores the resulting array in `r2`.

For each label `L` $\in$ `labels`, the instruction `goto L` transfers control to the label-instruction `L`, which must be unique in each program. The instruction `br L` transfers control to `L` only if an implicit Boolean *condition register* $\mathtt{r}_c$ stores True.

For each bitwise-arithmetic operation `op` $\in \mathtt{ops}_A$ and all virtual registers `r0`, `r1`, and `r2`, the instruction `op r0,r1,r2` computes a bitwise-arithmetic operation corresponding to `op` on the values stored in `r0` and `r1` and stores the result in `r2`. The operations corresponding to each operation in $\mathtt{ops}_A$ are addition, subtraction, multiplication, division, bitwise-and, and bitwise-or, respectively.

For each comparitor `cmp` $\in \mathtt{ops}_C$ and all virtual registers `r0` and `r1`, the instruction `cmp r0, r1` evaluates `cmp` on the values stored in `r0` and `r1` and stores the result in an implicit Boolean register $\mathtt{r}_c$. The comparisons corresponding to each comparator in $\mathtt{ops}_C$ are equality, non-equality, less-than, greater-than, greater-than-or-equal, and less-than-or-equal, respectively. The instruction `neg` negates the value in $\mathtt{r}_c$.

For each label `fn` $\in$ `Labels` and all virtual registers `r`, `r0`, ..., `rn`, the instruction `callr r,fn,r0,...,rn` stores all current bindings to virtual registers on a callstack and transfers control to instruction `fn`. When a matching `return` is executed, the returned value is stored in `r`. For each virtual register `r`, the instruction `return r` returns control to the instruction following the last unmatched `call` instruction, returning the value stored in `r`.

1.2 A Code Generator

The second key component of the assigned compiler phase is a code generator that takes an AST of a well-typed Tiger program and generates an equivalent IL program. The design of one possible code generator for a target intermediate language similar to the language described in §1.1 was given in class.

1.3 An Interpreter for Intermediate Code

The third component of the assigned compiler phase is an interpreter for the intermediate language IL that you will define (§1.1). Unlike the first two components, the interpreter is not critical for implementing the later phases of your compiler. However, the interpreter will be critical for debugging the implementation of your intermediate language and code generator.

One natural way to implement the interpreter is as a loop that maintains the current state of the program. In each iteration of the loop, the interpreter looks up the next instruction to be executed and updates the current state according to the semantics of the instruction.

Example code for an interpreter on which you may base your own is provided.

1.3.1 The State class

At the heart of the sample interpreter is the `State` class, which represents program state and provides operations for loading and storing values. State provides an abstract data-type that is a stack of frames, in which each frame roughly corresponds to the state of the currently executing function call. Conceptually, entering a function involves pushing a frame onto the stack, while exiting a function involves popping a frame from the stack. Care should be taken to save and restore appropriate state information when transitioning between two frames. Each frame is a key-value store in which the key is a String and the value is an Integer, Float, or String. A newly created State is initialized with a single frame called the global frame.

1.3.2 Manipulating state

The `State` class provides methods for inserting key-value pairs into the frame stack. These methods take as arguments: Boolean local, String key, and Object value. When local is true, inserts are made to the frame at the top of the stack. When local is false, insert are made to the global frame.

The `State` class also provides methods for retrieving the value associated with a key from the frame stack. These methods take the desired key then query the top frame for the corresponding value. If the query to the top frame fails, then the global frame is queried.

1.3.3 Arrays

The `State` class provides methods for allocating arrays, loading from arrays, and storing to arrays.

When an array is allocated, space for the content of the array is reserved in a special managed memory distinct from the frame stack. A key-value pair is then inserted into the frame stack. The key is the name of the array and the value is the array's base address in the managed memory as an integer.

Loading from and storing to an array involves querying the frame stack for the base address of the desired array. The base address and desired offset are added to determine the address of the desired array member in the managed memory.

1.3.4 Standard Library Functions

The `State` class provides implementations of standard library functions `readi`, `readf`, `printi`, and `printf`. You *must* use these implementations. Failure to do so may cause your interpreter to fail the tests used for grading.

1.3.5 Design

The interpreter for your intermediate language can be implemented as a simple client of the `State` class. In particular, the top-level loop of the interpreter may load values from a maintained `State` using the provided operations, perform computation on the values dependent on the current instruction, and store the result in an updated state using other `State` operations.

The `State` class can be used to emulate common constructs from execution environments you are familiar with. For example, use the global frame to model general purpose registers and the program counter. Insert the return address to which execution should resume after a function exits into the top frame.

The sample interpreter and test case demonstrate how to use the the `State` class. Do not take the sample interpreter as a definitive reference for your design. You may design your intermediate code format and interpreter however you choose.

2 Resources

To complete your work, you have access to the following resources:

- A set of test programs and test inputs for each program. Each input is given as a plaintext file, which can be redirected to the standard input of your compiler and interpreter.
- Sample code for a standalone interpreter, including the `State` class and `Memory` class. The classes have been designed so that you can use them as a client who relies only on their interface without relying on details of the internal representation or concrete implementation. However, you are free to read or even modify the implementations of any of the classes as you see fit. A tarball containing the source files for each class has been uploaded to T-Square.

3 Evaluation

Please extend your compiler to support a flag `--runil`. When the compiler is run with `--runil`, it should attempt to parse the input program, typecheck it, generate a representation in intermediate code, and then execute the program. As the program is run, it may read input from standard input and write outputs to standard output by invoking particular Tiger library functions.

To evaluate your compiler, we will run it on a set of Tiger programs. Each input Tiger program will be syntactically well-formed and well-typed, so if there are bugs in your compiler that are triggered by ill-formed or ill-typed programs, then you will not be penalized. However, if your compiler fails to parse or type-check a given program, then it will fail the test case. We will run your compiler on each input program with flag `--runil`, provide input to the interpreted program on standard input, collect the output that the program writes to standard output, and compare it to the output generated by our reference implementation. We will award points based on each test program and test input on which your interpreter matches the output of the reference implementation.

4 Collaboration

We will award identical grades to each member of a given project team, unless members of the team directly register a formal complaint. We assume that the work submitted by each team is their work solely. Any non-obvious discussion or questions about design and implementation should be either posted on the course's Piazza message boards privately for the instructors or presented in person during office hours or after lecture. If the instructors determine that parts of the discussion are appropriate for the entire class, then they will forward selections. Under no condition is it acceptable to use code written by another team. The instructors will automatically check each submitted solution against other solutions submitted and against other known implementations.