编译器构造实验(4): 代码生成

May 20, 2021

1. Code Generation

In this project, you are to write a code generator, the final phase of your compiler.

2. Due Date [截止时间]

The assignment is due June 17, 2021 23:59.

3. Project Summary［总体任务］

Your task is to write a code generator, which produces (target) assembly code for the MIPS R2000 architecture. It takes as input the augmented AST and symbol table produced by the previous phases of your compiler. The generated code will be executed using SPIM S20, a simulator for the MIPS R2000.

Code generation will consist of assigning memory addresses for each variable used in the MINIJAVA program and translating subtrees of the AST into sequences of assembly instructions that perform the same task.

4. Generating Code [生成代码]

This is the only phase of your compiler which is machine dependent. The important/interesting issues in generating code for MINI-JAVA are discussed in the following paragraphs.

Examples of assembly code programs are provided. You can make the following assumptions to simplify the project.

- Code should be generated for AST nodes on a node-by-node basis without taking into account the context of the node (i.e. the code should work regardless of where that node occurs in the AST).
- Code should be generated for the AST nodes in the order that they occur within the AST when doing depth-first left-right traversal (i.e. program order).
- You may take advantage of any special instruction in the machine when choosing target instructions to implement a given node.
- You do not have to do any fancy register allocation or optimization on your target code. (i.e. your code can be inefficient as long as it is correct.)

• Computing Memory Addresses [计算内存地址]

Since address information has not been computed and entered into the symbol table by earlier phases, the first task of the code generator is to compute the offsets of each variable name (both global and local); that is, the address of each local data object and formal parameter within the activation record

for the block in which they are declared. This can be done by initializing a variable **offset** at the start of each declaration section, and as each declaration is processed, the current value of **offset** is entered as an attribute of that symbol, and **offset** is then incremented by the total width of that data object (depending on its type).

The program execution begins with a method main() being called. All data for classes are instantiated when program execution begins. Thus, all storage for all classes is allocated statically in data sections inlined in code. The offsets of variables within classes should be computed and stored as an attribute of the variable name, typically relative to the start of the class.

For simplicity, declarations within a method do not contain any objects whose types are classes. That is, local variables can only be of integer type or integer array type.

Call-by-value parameters will have a width dependent on the type of the parameter (remember we are using only integer parameters), whereas call-by-reference parameters will have a width equal to ONE word to store an address.

The machine architecture must be take into account when computing these widths, that is, an integer in the MIPS processor is 4 bytes. Offsets of locals can be implemented as a negative offset from the frame pointer while offsets of parameters can be positive from the frame pointer. Thus, the computation of offsets of arguments and local variables can be done independently of each other. These offsets will be used for generating code for every reference to variables, in addition to being used in the allocation of storage for activation records.

- **Handling Structure Data Types**［处理结构数据类型］

Storage for an array is allocated as a consecutive block of memory. Access to individual elements of an array is handled by generating an address calculation using the base address of the array, the index of the desired element, and the size of the elements. You are free to choose the layout of elements of an array in your implementation (e.g., row major or column major order).

- **Simple Control Flow**［简单控制流］

Code for simple control statements, namely **if**s and **loop**s in MINI-JAVA, can be generated according to the semantics of conventional programming languages using the compare and branch instructions of the assembly language. Unique target labels will also have to be generated. You do not need to implement short circuiting when computing boolean expressions and just evaluate the entire expression.

- **Method Invocation**［函数调用］

Recursion in MINI-JAVA prevents the use of a static storage allocation strategy. However, the language has no features that prevent the deallocation of activation records in a last-in-first-out manner. That is, activation records containing data local to an execution of a method, actual parameters, saved machine status, and other information needed to manage the activation of the method can be stored on a run-time stack. The MIPS assembly language provides the subroutine call mechanisms to manipulate the run-time user stack.

An activation record for a method is pushed onto the stack upon a call to that method, while the activation record for the method is popped from the stack when execution of the method is finished

and control is to be returned to the caller. As MINI-JAVA does not allow dynamic classes and arrays, the sizes of all activation records are known at compile time.

Method calls result in the generation of a calling sequence. Upon a call, an activation record for the callee must be set up and control must be transferred to the callee after saving the appropriate information in the activation record. For each method, the generated code sequence will consist of a prologue, the code for the statements of the method, and the epilogue. Typically, the prologue saves the registers upon a method call and allocates space on the stack for local variables, whereas the epilogue consists of restoring the saved machine status and returning control to the point in the caller immediately after the point of call. The SPIM manual on the course website explains the instructions used to implement these actions.

- **Parameter Passing**［参数传递］

In order to correctly handle the formal parameters within the body of the callee, the symbol table entry for each formal parameter must include an attribute that indicates the parameter passing mode, that is, by-value or by-reference. Remember that we do not pass arrays as parameters. On a method invocation, call-by-value parameters are handled by allocating the local store for the size of the object in the activation record of the callee and then evaluating the actual parameter and initializing the local store within the callee with the value of the actual parameter. All accesses to that formal parameter will change the value in the local space, with no effect on the caller. On a return, no values are copied back to the caller.

Call-by-reference parameters are handled by allocating local space in the callee's activation record for the address of the actual parameter and then copying the address of the actual parameter into that local space. All accesses to that formal parameter during execution of the callee are indirect accesses through this address, having a direct effect on the caller. On return, no action is taken other than reclaiming the space.

Note that MIPS has a convention that the first 4 arguments of a method call are passed in register $a0-$a3. You have the option to follow this convention.

- **Register Usage**［寄存器使用］

In the MIPS processor, certain registers are typically reserved for special purposes. You should abide by these conventions.

- **Possible Functions**

You may want to write the following functions to help in the code generation.

1. emit_call(func name, num arg)/**emit a call instruction; func name: function id lexeme pointer; num arg: number of arguments */*
2. emit_label(l num)/**emit a definition of a label; l _num: label number; example: L=102, code generated = "L _102" */*
3. emit_goto(operator, l _num) /**emit unconditional and conditional jump instructions; operator: an operator in the branch-jump group; l num: label number */*

4. emit_data(name, type, size) /* *emit one data line, which is used for STATIC allocation; name: data object id lexeme pointer; type: type width; size: number of elements of above type* */
5. emit_str(name, str) /* *emit a string constant definition line; name: pointer to the name lexeme; str: pointer to the str* */
6. emit_most(operator, type, num op, op1, op2, op3) /* *emit most of the instructions; operator: one of the instructions in the general group; type: data type; num _op: number of operands 1, 2 and/or 3; op1...3: operands, op2 and op3 can be omitted depending on num _op* */

- **Run-time Error Detection**

You do not need to generate any run-time checks. Thus you can assume that array bounds are within range and scalars are within range.

5. Testing Code［测试代码］

The code generated is MIPS assembly code and should follow the descriptions specified in the handout SPIM S20. See the provided samples of generated assembly code.

Please note: the example code is by no means the only way to generate code. This is only an example and you can come up with your own generation schemes. In fact, no two compilers generate the same assembly for a given source code.

You can run the generated assembly code on the simulator and check the results. However, correct output does not guarantee that your code is completely correct. You should examine your generated code carefully.

```
$codeGen < sample1.java
$spim.linux -asm -file code.s
... ...
or
$spim.linux -asm
 (spim) load “code.s”
 (spim) step
 [0x00400000] 0x8fa40000 lw $4,0($29)
 (spim) run
... ...
```

* For the above to work, the file *trap.handler* must be in your working directory.
* Read the SPIM debugger manual to learn more about debugger commands

6. Assignment submission [关于提交]

The submission should be a compressed file that contains your project source code and report (no executable please).