CS39003 Compilers Laboratory, Autumn 2024–2025
Assignment No: 6
Date: 04-Nov-2024

Target-Code Generation

Language

In this assignment, you are required to generate assembly-level codes for a tiny version of a LISP-like language. The grammar for this language is given below. Variable assignments, expressions, and if (not if-else) and while statements are only to be supported. If statements are written by the keyword "when", and while statements are written by the keyword "loop while". The start symbol in the grammar below is LIST. Terminal symbols are highlighted in red. For simplicity, assume that all numbers are (signed) integers. For identifiers, use the same convention as in C.

LIST	⟶	STMT | STMT LIST
STMT	⟶	ASGN | COND | LOOP
ASGN	⟶	**(** **set** IDEN ATOM **)**
COND	⟶	**(** **when** BOOL LIST **)**
LOOP	⟶	**(** **loop** **while** BOOL LIST **)**
EXPR	⟶	**(** OPER ATOM ATOM **)**
BOOL	⟶	**(** RELN ATOM ATOM **)**
ATOM	⟶	**IDEN** | **NUMB** | EXPR
OPER	⟶	**+** | **–** | ***** | **/** | **%**
RELN	⟶	**=** | **/=** | **<** | **>** | **<=** | **>=**

Target assembly code

The target code will consist of the following types of assembly instructions.

LD R M	Load memory location M to register R.
LDI R C	Load integer constant C to register R.
ST M R	Store register R to memory location M.
OP R A B	Arithmetic operator OP is applied to arguments A and B, and the result is stored in register R. Here, A and B can be other registers or integer constants (but not memory locations).
JMP L	Unconditional jump to instruction number L.
JCND A B L	Conditional jump to label L (if A CND B is true, goto instruction number L). A and B can be registers or integer constants. CND can be EQ, NE, LT, LE, GT, GE. You do not need to support Boolean operators.

Parsing

Write a Lex file *prog.l* for extracting the tokens from the input. Write a Yacc/Bison file *prog.y* for parsing and for generating the intermediate three-address code. The parser will not prepare any parse tree. It will instead generate a table for quads (call this table **Q**) for storing the three-address instructions. Instruction numbers start from 1. Jump labels will be instruction numbers. Use incremental code generation using backpatching. Add suitable marker terminals to the grammar for the purpose of backpatching. For this simplified grammar without Boolean connectives, truelist, falselist, and nextlist can only be empty or singleton sets. That is, you do not need to maintain any "list" for backpatching. Moreover, merging is also not needed. Avoid redundant labels and jumps by using iffalse instead of if. Write all the parsing functions (including generation of the table **Q** of quads) in the Yacc file itself. You need to maintain a symbol table that will be needed in the target-code generation phase.

The rest of the work is done by a C/C++ file *codegen.c*(*pp*). This file should contain the main() function.

Processing the intermediate code

The program *codegen.c* starts by identifying the leaders and generating the basic blocks in the intermediate code. Print the blocks with line numbers, as demonstrated in the sample output. Labels are instruction numbers.

Target-code generation

Use the simple code-generation algorithm given in the Dragon Book. In the symbol table, store all the locations of (user-defined) variables. If you choose, you may also store all the temporaries in the symbol table. To avoid naming conflicts, store the temporaries as $1, $2, $3, and so on. Furthermore, maintain a data structure to store which variables and temporaries are stored in each register. Assume that the default number of registers is 5 (call them R1, R2, R3, R4, R5). You may run *codegen* with a command-line argument that overrides this default value to a value specified by the user at runtime. The target code will also be a sequence of three-address assembly instructions, and can be stored in another table of quads (call this table **T**).

The essential components of the simple code-generation algorithm are now briefly explained. For details, see the Dragon Book.

- Process the intermediate code **Q** block by block. Do not go for any treatment of interaction among different blocks. Also, there is no necessity to perform any optimization. Process the three-address codes (stored in **Q**) sequentially one by one, and store the relevant sequence of assembly instructions in **T**. The jump labels in the target code are again instruction numbers, and are in general different from the corresponding instruction numbers in the intermediate code.

- **Beginning of block:** Set all registers to be free.

- **Register allocation:** This is the most critical part in the target-code generation algorithm. Depending on the type of the three-address code that you are converting to the target code, the register-allocation algorithm changes.

Arithmetic operations

These operations are of the form OP T A B, where T is a temporary, and A and B are variables or temporaries or integer constants. There is no need to use a register for a constant argument. A non-constant argument must reside in a register for OP to take place. The register-allocation algorithm for A is now explained (that for B is similar). Apply the following sequence of attempts until a register is available for A.

1. If A is already residing in a register, use that register. There is no need to load A again from memory.
2. If there is some free register, load A to that register.
3. If there is a register such that all the variables and temporaries stored in that register have latest values in memory, load A to that register, after deleting that register from the storage locations of all those variables and temporaries.
4. If T and A are the same, and only T is stored in some register, use that register for A. Load A to that register, and use the same register for T.
5. If there is a register containing only temporaries, neither of which will be used in the rest of the block, load A to that register after freeing that register.
6. The score of a register is the number of variables and temporaries stored in that register, whose latest values are not available in memory. Find a register with the least score. For that register, save all unsaved variables and temporaries to memory. Free that register, and load A to that register.

For choosing a register for T (if not already set by Attempt 4 above), use the same sequence of attempts (except 4). Do not store T to memory (for variables), but record that the only location for T is the register allocated to T.

Set operations

A set instruction is of the form T = A. If A is a constant, choose a register for T by making the above attempts (except 4), and load that register with that constant value. Also record the only location of T to be that register. Do not issue a store instruction at this point (even if T is a variable).

If A is a variable or a temporary, we have a situation similar to an arithmetic operation with only one argument. Now, you use the same register for both T and A.

Conditional jumps

This is of the form iffalse A COND B goto L. This case is again similar to the case of an arithmetic operation with the exception that we do not need a target register to store the outcome of the condition check.

Unconditional jumps

No register allocation is necessary.

- **End of block:** Identify all variables that have been modified in the block and stored in registers, and are not written back to the memory. Issue store instructions for all these variables. Assume that no temporary is used across multiple blocks, so there is no need to write back the temporaries to the memory.

Print the assembly code

From the table **T**, print the assembly instructions in the sequence they are generated. Separate the target codes for the blocks by empty lines. Although the assembly code is not expected to have line numbers, print those too for clarity of your output. Moreover, assembly codes are expected to use offsets for memory references instead of variable or temporary names. For readability of your output, print the names. In reality, symbol table(s) store(s) offsets of variables (and temporaries), so it is straightforward to locate items in the symbol table(s) and replace the names by the respective offsets.

Makefile

Write a makefile with all, run, and clean targets. The target all will generate lex and yacc outputs, and compile *codegen.c*(*pp*) to an executable *codegen*. The target run will run *codegen* with stdin redirected from *sample.txt* (a sample is supplied on the next page). The clean target would clean all files generated by make all and run.

Submit

Prepare a single tar/tgz/zip archive containing *prog.l*, *prog.y*, codegen.c(pp*)*, *makefile*, and *sample.txt*. Submit that archive.

Sample

Input

```
(set x 2024)
(set n 0)
(loop while (/= x 1)
   (when (= (% x 2) 1)
      (set x (+ (* 3 x) 1))
      (set n (+ n 1))
   )
   (set x (/ x 2))
   (set n (+ n 1))
)
(when (= n 0) (set F 0))
(when (= n 1) (set F 1))
(when (>= n 2)
   (set m 2)
   (set Fprev1 1)
   (set Fprev2 0)
   (loop while (<= m n)
      (set F (+ Fprev1 Fprev2))
      (set Fprev2 Fprev1)
      (set Fprev1 F)
      (set m (+ m 1))
   )
)
```

Blocks of intermediate code

```
Block 1
   1    : x = 2024
   2    : n = 0

Block 2
   3    : iffalse (x != 1) goto 16

Block 3
   4    : $1 = x % 2
   5    : iffalse ($1 == 1) goto 11

Block 4
   6    : $2 = 3 * x
   7    : $3 = $2 + 1
   8    : x = $3
   9    : $4 = n + 1
   10   : n = $4

Block 5
   11   : $5 = x / 2
   12   : x = $5
   13   : $6 = n + 1
   14   : n = $6
   15   : goto 3

Block 6
   16   : iffalse (n == 0) goto 18

Block 7
   17   : F = 0

Block 8
   18   : iffalse (n == 1) goto 20

Block 9
   19   : F = 1

Block 10
   20   : iffalse (n >= 2) goto 32

Block 11
   21   : m = 2
   22   : Fprev1 = 1
   23   : Fprev2 = 0

Block 12
   24   : iffalse (m <= n) goto 32

Block 13
   25   : $7 = Fprev1 + Fprev2
   26   : F = $7
   27   : Fprev2 = Fprev1
   28   : Fprev1 = F
   29   : $8 = m + 1
   30   : m = $8
   31   : goto 24

   32   :
```

Target code

```
Block 1
   1    : LDI R1 2024
   2    : LDI R2 0
   3    : ST x R1
   4    : ST n R2

Block 2
   5    : LD R1 x
   6    : JEQ R1 1 24

Block 3
   7    : LD R1 x
   8    : REM R2 R1 2
   9    : JNE R2 1 17

Block 4
   10   : LD R1 x
   11   : MUL R2 3 R1
   12   : ADD R3 R2 1
   13   : LD R1 n
   14   : ADD R4 R1 1
   15   : ST x R3
   16   : ST n R4

Block 5
   17   : LD R1 x
   18   : DIV R2 R1 2
   19   : LD R1 n
   20   : ADD R3 R1 1
   21   : ST x R2
   22   : ST n R3
   23   : JMP 5

Block 6
   24   : LD R1 n
   25   : JNE R1 0 28

Block 7
   26   : LDI R1 0
   27   : ST F R1

Block 8
   28   : LD R1 n
   29   : JNE R1 1 32

Block 9
   30   : LDI R1 1
   31   : ST F R1

Block 10
   32   : LD R1 n
   33   : JLT R1 2 53

Block 11
   34   : LDI R1 2
   35   : LDI R2 1
   36   : LDI R3 0
   37   : ST m R1
   38   : ST Fprev1 R2
   39   : ST Fprev2 R3

Block 12
   40   : LD R1 m
   41   : LD R2 n
   42   : JGT R1 R2 53

Block 13
   43   : LD R1 Fprev1
   44   : LD R2 Fprev2
   45   : ADD R3 R1 R2
   46   : LD R2 m
   47   : ADD R4 R2 1
   48   : ST F R3
   49   : ST Fprev2 R1
   50   : ST Fprev1 R3
   51   : ST m R4
   52   : JMP 40

   53   :
```