CSc 453 — Assignment 1

Christian Collberg
University of Arizona, Department of Computer Science

January 18, 2026

1 Introduction

Your task is to extend the TINY language described in class with if-statements, goto-statements, and labels. The purpose of this assignment is to get you introduced to the inner workings of a compiler and the different translation phases a program has to go through before it can be executed.

2 Getting Started

This assignment can be done in teams of one or two.

To get started, follow these instructions:

1. Unzip `assignment-1.zip` which gives you these files:

   ```
   assignment-1/assignment-1.pdf
   assignment-1/validator.sh
   assignment-1/firstname_lastname/tiny-source/*.java
   assignment-1/firstname_lastname/tiny-source/*.tny
   assignment-1/firstname_lastname/tiny-source/Makefile
   assignment-1/firstname_lastname/collaboration.json
   assignment-1/firstname_lastname/survey.json
   assignment-1/firstname_lastname/fib.tny
   assignment-1/firstname_lastname/validator.sh
   ```

2. Then, assuming your name is Pat Jones, execute these commands:

   ```
   > cd assignment-1
   > mv firstname_lastname pat_jones
   ```

   If you are working in a group of two, pick one of your names.

3. Now, go ahead and solve the problems, entering your answers in the template files in `pat_jones/*`. Your main work should be modifying the Java source files in `assignment-1/pat_jones/tiny-source`.

Don't show your code to anyone outside your team, don't read anyone else's code, don't discuss the details of your code with anyone. If you need help with the assignment see the TA or the instructor.

3 Submission

To submit, follow these instructions:

1. Verify that your submission contains a working `Makefile` and *all* the files necessary to build the compiler. Your program must compile out of the box. The graders will *not* try to debug your program or your makefile for you.
2. Fill out `collaboration.json` with information about your group.
3. Optionally fill out `survey.json` to give me feedback about the assignment.
4. Package up your answers and verify that they look OK:

```
> zip -r pat_jones.zip pat_jones
> assignment-1/validator.sh pat_jones.zip
==== This is the LigerLabs assignment validator ====
          ...        ...        ...        ...
==== 6: Checking that all problems have been answered.
==== 7: Checking that all answer files have valid structure.
```

5. Upload `pat_jones.zip` to d2l. If you are working in a group of two, you should only submit once.

4 The Tiny Language

Here's a simple Tiny program `mul.tny` that prints out the product of two numbers held in variables `x` and `y`:

```
1   BEGIN
2      x = 10;
3      y = 20;
4      S = 1;
5
6      IF 0 < x GOTO 22;
7      S = 0;
8      x = 0 - x;
9      22:;
10
11     P = 0;
12     11:;
13     IF x < 0 GOTO 99;
14     P = P + y;
15     x = x - 1;
16     GOTO 11;
17
18     99:;
19     IF S GOTO 88;
20     P = 0 - P;
21
22     88:;
23     PRINT P;
24  END
```

Note that TINY doesn't have loops or structured if-statements. Rather, it has labels (consisting of a number followed by a colon), and unconditional branches (⌜`GOTO label`⌝) and conditional branches (⌜`IF expr GOTO label`⌝) that can jump to these labels. Also, it only has three arithmetic operators: `+`, `-`, and `<`. The `<`-operator compares its two arguments and produces 1 if the left hand side is less than the right hand side, and 0 otherwise. The if-statement ⌜`IF expr GOTO label`⌝ will branch to `label` if the `expr` is not equal to 0.[1]

Here is the complete concrete grammar:

program	→	'BEGIN' stats 'END'
stats	→	stat stats | ϵ
stat	→	ident '=' expr ';'
	|	'PRINT' expr ';'
	|	'IF' expr 'GOTO' int';'
	|	'GOTO' int ';'
	|	int ':' ';'
expr	→	expr '+' expr
	|	expr '-' expr
	|	expr '<' expr
	|	ident
	|	int
ident	→	LETTER idp
idp	→	LETTER idp | DIGIT idp | ϵ
int	→	DIGIT intp
intp	→	DIGIT intp | ϵ

5 Translating and Running TINY Programs

The `TinyC` compiler for the TINY language generates Mips code for the QTSpim simulator:

```
> java Compiler mul.tny > mul.as
> cat mul.as
        .data
newline:        .asciiz "\n"
var0:           .word 0
var1:           .word 0
var2:           .word 0
var3:           .word 0
        .text
        .align 2
        .globl main
main:
        li      $s0,10
        sw      $s0,var0
        li      $s0,20
        sw      $s0,var1
```

[1] TINY is not unlike early versions of the BASIC language.

```
        li      $s0,1
        sw      $s0,var2
        li      $s0,0
        lw      $s1,var0
        sle     $s2,$s0,$s1
        bnez    $s2, L22
        li      $s3,0
        sw      $s3,var2
        li      $s0,0
        lw      $s1,var0
        sub     $s2,$s0,$s1
        sw      $s2,var0
L22:
        li      $s0,0
        sw      $s0,var3
L11:
        lw      $s0,var0
        li      $s1,0
        sle     $s2,$s0,$s1
        bnez    $s2, L99
        lw      $s3,var3
        lw      $s4,var1
        add     $s5,$s3,$s4
        sw      $s5,var3
        lw      $s0,var0
        li      $s1,1
        sub     $s2,$s0,$s1
        sw      $s2,var0
        b       L11
L99:
        lw      $s0,var2
        bnez    $s0, L88
        li      $s1,0
        lw      $s2,var3
        sub     $s3,$s1,$s2
        sw      $s3,var3
L88:
        lw      $s0,var3
        move    $a0,$s0
        li      $v0,1
        syscall
        la      $a0,newline
        li      $v0,4
        syscall
        li      $v0,10
        syscall
```

`TinyC` can also execute the program by traversing the abstract syntax tree (`Eval.java`) or interpreting stack code (`Interpreter.java`):

```
java Eval 4.tny
java GenIR 4.tny > 4.ir
java Interpreter 4.ir
```

6 The TinyC Compiler

The compiler consists of these phases and classes:

- lexical analysis (`Token`, `Lex`),
- syntactic analysis (`Parse`),
- abstract syntax tree construction (`Parse`, `AST`, `PROGRAM`, `STAT`, `STATSEQ`, `ASSIGN`, `PRINT`, `EXPR`, `NULL`, `IDENT`, `INTLIT`, `BINOP`),
- semantic analysis (`SyTab`, `Sem`),
- intermediate code generation (`IR`, `GenIR`),
- machine-code generation (`GenMips`),
- interpretation (`Interpreter`),
- the `Compiler` class ties it all together:

```
1   public class Compiler {
2       public static void main (String args[]) throws IOException {
3           String file = args[0];
4           Lex scanner = new Lex(file);         // Lexical Analysis
5           Parse parser = new Parse(scanner);   // Syntactic Analysis
6           Sem sem = new Sem(parser.ast);       // Semantic Analysis
7           GenIR ir = new GenIR(sem);           // Intermediate Code Generation
8           GenMips mips = new GenMips(ir.code); // MIPS Code Generation
9           System.out.println(mips.code);       // Print generated code
10      }
11  }
```

7 Problem 1

The version of the compiler you have been given does not implement `IF`, `GOTO`, labels, or the `-` and `<` operators. It is your task to add these to the compiler and interpreter. More specifically, you should do the following:

1. Modify `Token.java` by adding definitions of the new tokens in the language: `:`, `IF`, `-`, `<`, `GOTO`.
2. Modify `Lex.java` to handle the new tokens.
3. Add new classes representing AST nodes for the `IF`-, `GOTO`- and `label`-statements. Look at `ASSIGN.java` to get an idea of what these should look like.
4. Modify `Parse.java` to parse `IF`-, `GOTO`- and `label`-statements[2] and to build AST nodes for these. `Parse.java` should also handle the new operators `-` and `<`. Note that all TINY operators have the same precedence, i.e. expressions are evaluated strictly left-to-right.
5. Modify `Sem.java` to do semantic analysis for the new statements. The only semantic rules of interest are:
   - A label can only be declared once.

[2] In this language labels are a kind of statement. This is ugly (you have to put a semicolon after each label), but it simplifies the compiler.

- A target of a branch (`label` in ⌜`IF expr GOTO label`⌝ or ⌜`GOTO label`⌝) has to be defined. Forward jumps are OK, however.

6. Add new instructions to the intermediate code in `IR.java`. Exactly how you do this is up to you, but you might want to add the intermediate code statements `SUB`, `LT`, `BRA`, and `BRNEO`:

mnemonic	stack-pre	stack-post	side-effects
ADD	[A,B]	[A+B]	
SUB	[A,B]	[A-B]	
LT	[A,B]	if A<B then 1 else 0	
BRA X	[]	[]	Branch to label X
BRNEO X	[A]	[]	If A is not equal to 0 branch to label X
LOAD X	[]	[Memory[X]]	
STORE X	[A]	[]	Memory[X] = A
PUSH X	[]	[X]	
PRINT	[A]	[]	Print A
PRINTLN	[]	[]	Print a newline
HEADER M,V	[]	[]	
EXIT	[]	[]	The interpreter exits

7. Modify `Interpreter.java` to handle the new intermediate code instructions.

8. Modify `GenMips.java` to handle the new intermediate code instructions.

8 Problem 2

Implement the Fibonacci function in TINY. Name the program `fib.tny`.