Project 2
CISC 471 Compiler Design
A Syntax-directed Translator for a Small Imperative Language (SIL-K)

Deliverables and Deadlines: **April 30 first deliverable.**
May 6 second deliverable. See below for more details.

You may work by yourself or in pairs.

Project Objectives: **To gain experience writing a syntax-directed translator and learn how to read and modify someone else's software.**

Procedure:

Write an SDT (syntax-directed translator) to generate code for the simple imperative language shown below, called SIL-K. The SIL-K language does not contain any procedures, but only a single main program. Base types are limited to integer only. Arrays are one-dimensional (or 2-dimensional for extra credit!) with the integer type as its component and index type. The following statements are included: for-do, if-then, if-then-else, assignment, write, and compound statement. Operators are restricted to arithmetic and relational. The grammar that we are using for the SIL-K language is as follows:

```
start     ::=   program ID ; block .

block     ::=   variables cmpdstmt

variables        ::=   var vardcls | empty string

vardcls          ::=   vardcls vardcl ; | vardcl ;

vardcl    ::=   IDlist : type

type      ::=   integer | array[ ICONST ] of integer | array[ ICONST, ICONST ] of
integer

IDlist    ::=   IDlist , ID | ID

stmtlist         ::=   stmtlist ; stmt | stmt

stmt      ::=   ifstmt | fstmt | astmt | writestmt | cmpdstmt

cmpdstmt         ::=   begin stmtlist end

writestmt        ::=   writeln ( exp )

ifstmt    ::=   ifhead then stmt else stmt | ifhead then stmt
```

```
ifhead  ::=     if condexp

fstmt   ::=     for ctrlexp do stmt

ctrlexp         ::=     ID := ICONST, ICONST

astmt   ::=     lhs := exp

lhs     ::=     ID | ID [ exp ] | ID [ exp, exp ]

exp     ::=     exp + exp | exp - exp | exp * exp | ID | ID [ exp ] | ID [ exp, exp ] |
ICONST

condexp         ::=     exp != exp | exp == exp | exp < exp | exp <= exp
```

You may assume that the SIL-K programs are correct in terms of static semantics, i.e., no semantic analysis (type checking) is required.

You will write a syntax-directed translation scheme that will generate ILOC code for the above language. ILOC is an intermediate language that is similar to assembly code. You may test the correctness of your generated ILOC code by running it on the ILOC simulator sim provided in directory /usa/Pollock/cis471/project2 on orioles. This directory also contains the source code of the ILOC simulator if you want to compile it on another machine for testing.

Code Shape Requirements

* Your code should use the register-register model that exposes the maximal opportunities for register allocation. In other words, each new value should reside in a separate virtual register. The function NextRegister will return a new (fresh) register number each time it is called.

* The first element of an array a is a[0] (one-dimensional) or a[0,0] (two-dimensional). All addresses are byte addresses and an integer value is stored in a 4 byte word. The data layout for two-dimensional arrays should be column-major order. The overall available memory is set to 20,000 bytes. For instance, if you specify array x[100,100] of integer , the simulator will complain! Again, 2-dimensional array translation is extra credit, and not required.

* All variables are statically allocated, i.e., there is no need for activation records on a runtime stack. The static area starts at memory location 1024. Addresses above are reserved for register spilling. The register r0 should contain the starting address (namely 1024) of the static area during program execution.

* You may only use the following ILOC instructions. All these instructions are implemented in sim , our ILOC simulator. A table of ILOC instructions and their semantics is given at the end of this document.

- no operation: nop .

- arithmetic: addI, add, subI, sub, mult .

- memory load, loadI, loadAO, loadAI, store, storeAO, storeAI .

- control flow: br, cbr, cmp_LT, cmp_LE, cmp_EQ, cmp_NE, cmp_GT, cmp_GE .

- I/O: output .

Please see files instrutil.h and instrutil.c for the definitions of procedures/functions emit, emitComment, NextRegister, and NextLabel.

* You may want to generate nop instructions as targets of branches and conditional branches, e.g., L1: nop .

* The evaluation of an exp will always result in an integer value, while the evaluation of a condexp will always result in a boolean (0 or 1) value. An ILOC cmp_ instruction writes a boolean value into its target register.

* The function NextLabel will generate a new (fresh) label each time it is called.

How To Get Started

The following code is provided as a starting point for your project. You can copy the files from the directory /usa/Pollock/cis471/project2 on orioles.

1. Scanner: scan.l (flex) *** DO NOT MODIFY ***

2. Parser/Code Generator: parse.y (bison). Here is where most of your code will go. It contains an example of how to use procedure emit to generate code. You will need to remove this in your final version, i.e., it has only be inserted as an illustration example.

3. attr.h and attr.c . You will need to define new attribute(s) for recording information as you parse to use it later in parsing for generating code.

4. symtab.h and symtab.c . May need to be modified.

5. instrutil.h and instrutil.c . Need not modify.

6. Makefile Need not modify.

In order to get started on testing your compiler, you can use the following test cases. This is just a tentative list of source codes and their generated sample ILOC code using our code generator sample solution (codegen found in /usa/pollock/cis471/project2 on orioles). We will use many more test cases to grade your project . There are many ways of generating correct code, so our codegen compiler gives you only an overall idea what needs to be done.

1. Basic straight line (no conditionals or loops) code:

    * demo1 (demo1.out)

2. Basic code with control flow:

    * demo2 (demo2.out)

    * demo3 (demo3.out)

3. Basic code with control flow and array references:

    * demo4 (demo4.out)

    * demo5 (demo5.out)

    * demo6 (demo6.out)

4. More code with control flow and array references:

    * demo7 (demo7.out)

    * demo8 (demo8.out)

You can generate an executable called codegen by typing make. The parser/code generator expects the input on stdin, i.e., you can call the parser on an input program as follows: codegen < demo1. The parser/code generator writes the resulting ILOC code into file iloc.out.

Submission Procedure

You can either tar all of your files and email them to the ta by the deadline, or let the ta know what subdirectory of your svn has your files and that it is ready for grading. If you email the files, please tar all your source files, including the ReadMe file. Your ReadMe file may contain comments that you want the grader to know about. Do a "make clean" before tar-ing your files. Do not submit your compiler as an executable. Do not submit the simulator or the provided sample solution.

Grading Criteria

The project will be mainly graded on functionality. You will receive no credit for the entire project if we cannot recreate (make) your compiler or your compiler does not run on any of our test codes.

Grading rubric: 65 points
25 points: Basic points for basic code generation for the following features: ***due April 30***

5 points: Start, block, integer variables

5 points: arithmetic expressions

5 points: assignment

5 points: write statement

5 points: sequences of statements

Control features: ***complete code generator due May 6***

10 points: Conditional expressions, if statements (then and then/else)

8 points: For (loop) statement

7 points: Single dimensional array

5 points: nested if statements

5 points: nested loops

5 points: nested loops and if statements combined

Extra Credit: 10 points 2-dimensional arrays ***due May 6***

Credits: This project was created by Uli Kremer, an Assistant Professor at Rutgers University. He has given permission for this project to be used in this compiler course.

ILOC Instruction Set

Operation				Meaning
loadI	c	$\Rightarrow$	r_x	c $\rightarrow r_x$
load	r_x	$\Rightarrow$	r_y	MEM(r_x) $\rightarrow r_y$
loadAI	r_x, c_y	$\Rightarrow$	r_z	MEM($r_x + c_y$) $\rightarrow r_z$
loadAO	r_x, r_y	$\Rightarrow$	r_z	MEM($r_x + r_y$) $\rightarrow r_z$
store	r_x	$\Rightarrow$	r_y	$r_x \rightarrow$ MEM(r_y)
storeAI	r_x	$\Rightarrow$	r_y, c_z	$r_x \rightarrow$ MEM($r_y + c_z$)
storeAO	r_x	$\Rightarrow$	r_y, r_z	$r_x \rightarrow$ MEM($r_y + r_z$)
nop				no operation
addI	r_x, c	$\Rightarrow$	r_z	r_x + c $\rightarrow r_z$
add	r_x, r_y	$\Rightarrow$	r_z	$r_x + r_y \rightarrow r_z$
subI	r_x, c	$\Rightarrow$	r_z	r_x - c $\rightarrow r_z$
sub	r_x, r_y	$\Rightarrow$	r_z	r_x - $r_y \rightarrow r_z$
mult	r_x, r_y	$\Rightarrow$	r_z	$r_x * r_y \rightarrow r_z$
br		$\Rightarrow$	L_x	$L_x \rightarrow$ PC (program counter)
cbr	r_x	$\Rightarrow$	L_y, L_z	if r_x = true, then $L_y \rightarrow$ PC
				if r_x = false, then $L_z \rightarrow$ PC
cmp_LT	r_x, r_y	$\Rightarrow$	r_z	if $r_x < r_y$ then **true** $\rightarrow r_z$
				otherwise **false** $\rightarrow r_z$
cmp_LE	r_x, r_y	$\Rightarrow$	r_z	if $r_x \leq r_y$ then **true** $\rightarrow r_z$
				otherwise **false** $\rightarrow r_z$
output	c			print MEM(c)

The same register can occur more than once in the same instruction. Examples: `load` r_1 `=>` r_1 and `add` r_1`,` $r2$ `=>` r_1. Instructions may have labels of the form L_x. Other control flow instructions such as `cmp_EQ` and `cmp_NE` follow the same definitions as the two control flow instructions listed in the table above.