Description

Compiler.c: A recursive descent LL(1) parser that generates RISC machine instructions.

InstrUtils.c : A utility function that prints the list of RISC machine instructions. A sequence of machine instructions is represented by a doubly linked list.

Optimizer.c: A peephole optimizer for constant propagation using a sliding window of three RISC machine instructions. It looks for a pattern of the following form:

```
LOADI Rx #c1

LOADI Ry #c2

op Rz Rx Ry
```

If this pattern is detected, the value of constants c1 op c2 is computed as constant c3, where op can be addition ADD, subtraction SUB, or multiplication MUL . The original sequence of three instructions is then replaced by a single instruction of the form:

```
LOADI Rz #c3
```

If no pattern is detected, the window is moved one instruction down the list of instructions. In the case of a successful match and code replacement, the first instruction of the new window is set to the instruction that immediately follows the three instructions of the pattern in the original, unoptimized code. Instructions that are deleted as part of the optimization process have to be explicitly deallocated using the C free command in order to avoid memory leaks.

Instructions

1. Extract the project zip file.
2. Type `make compile` to generate the compiler.
3. To run the compiler on testcase 1, type `./compile test1`. This will generate a RISC machine program `tinyL.out`.
4. To create the optimizer, say `make optimize`.
5. To call the optimizer on a file that contains RISC machine code, say tinyL.out, say `./optimize <tinyL.out > omptimized.out`.
6. The RISC virtual machine can be generated by saying `make run`.
7. To run a program on the virtual machine, say tinyL.out, say `./run tinyL.out`.
8. You can define a tinyL language interpreter on a single Linux command line as follows:
   `./compile test1; ./optimize < tinyL.out > opt.out; ./run opt.out`

The RISC **machine instruction set**

R_x , R_y , and R_z represent three arbitrary, but distinct registers.

instruction format	description	semantics
	memory instructions	
LOADI R_x #<const>	load constant value #<const> into register R_x	$R_x \leftarrow$ <const>
LOAD R_x <id>	load value of variable <id> into register R_x	$R_x \leftarrow$ <id>
STORE <id> R_x	store value of register R_x into variable <id>	<id> $\leftarrow R_x$
	arithmetic instructions	
ADD R_x R_y R_z	add contents of registers R_y and R_z , and store result into register R_x	$R_x \leftarrow R_y + R_z$
SUB R_x R_y R_z	subtract contents of register R_z from register R_x and store result into register R_x	$R_x \leftarrow R_y - R_z$
MUL R_x R_y R_z	multiply contents of registers R_y and R_z , and store result into register R_x	$R_x \leftarrow R_y \_ R_z$
	I/O Instructions	
READ <id>	read value of variable <id> from standard input	read(<id>)
WRITE <id>	write value of variable <id> to standard output	print(<id>)

The tinyL **language**

tinyL is a simple expression language that allows assignments and basic I/O operations.

<program>	::=	<stmt_list> .
<stmt_list>	::=	<stmt> <morestmts>
<morestmts>	::=	; <stmt list> | ε
<stmt>	::=	<assign> | <read> | <print>
<assign>	::=	<variable> = <expr>
<read>	::=	? <variable>
<print>	::=	! <variable>
<expr>	::=	+ <expr> <expr> |
		− <expr> <expr> |
		* <expr> <expr> |
		<variable> |
		<digit>
<variable>	::=	a | b | c | d | e
<digit>	::=	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Examples of valid **tinyL** programs:
?a;?b;c=+3*ab;d=+c1;!d.
?a;b=-*+1+2a58;!b.