CSE 302: Compiler Design — Lab 3

Control Structures

Out: `2019-09-26 13:15:00`
Tests due: `2019-10-03 15:15:00`
Compilers due: `2019-10-10 15:15:00`

---

1 INTRODUCTION

In this lab you will modify the compiler you built in lab 2 to support the language `BX1`, which is an extension of `BX0` with booleans, comparisons, and control structures.

- **Tests**: by `2019-10-03`, we would like you to submit 5 *non-trivial* test programs written in the `BX1` language. These programs will be made available to everyone during the second week.
- This lab is intended to be worked on individually, not in teams.

This lab will be assessed. It is worth 15% (i.e., 3 points) of your final grade. To get full credit, you must submit the final version of your compiler in 2 weeks, i.e., strictly before `2019-10-10 15:15:00 CEST`. Every additional hour past this deadline incurs a $(20/24)\% = 0.83\%$ penalty.

2 THE `BX1` LANGUAGE

2.1 *Syntax*

The `BX1` language is a strict superset of the `BX0` language, but not all valid `BX0` programs will be valid `BX1` programs. This is because `BX1` adds *variable declarations* and a new requirement that every variable that is used in the program must be declared at the top of the program.

Line Comments `BX1` allows for comments in the source text. Each comment begins with the literal text `//`, which causes the entire rest of the line (i.e., up to the next occurrence of a newline character or the end of the file) to be ignored. Note that C-style comments delimited between `/*` and `*/` are not supported. An entire line comment is interpreted as the equivalent of a single space character. Thus, the following code is syntactically equivalent to the ill-formed statement `y = x < < 10;`

```
y = x <// cannot break in the middle of a multi-character operator or keyword
< 10;
```

Types Unlike in `BX0` where all variables (and temporaries) were of integral type, in `BX1` there are two types: signed 64-bit integers (`int64`) and booleans (`bool`).[1] These two types are kept completely separate, so a boolean cannot be implicitly converted to an integer, nor can an integer be interpreted as a boolean. The boolean type has two new boolean-valued immediates (i.e., constants): `true` and `false`.

⟨type⟩ ::= `"int64"` | `"bool"`

[1] Stylistic note: we will keep the word "boolean" uncapitalized in this course.

operator	description	arity	precedence
`||`	boolean or	binary	3
`&&`	boolean and	binary	6
`|`	bitwise or	binary	10
`^`	bitwise xor	binary	20
`&`	bitwise and	binary	30
`==`, `!=`	equality	binary	33
`<`, `<=` `>`, `>=`	inequalities	binary	36
`<<`, `>>`	bitwise shifts	binary	40
`+`, `-`	addition, subtraction	binary	50
`*`, `/`, `%`	multiplication, division, modulus	binary	60
`-` , `!`	integer/boolean negations	unary	70
`~`	bitwise complement	unary	80

Figure 1: Operator precedence table for BX1 (higher precedence binds tighter)

VARIABLE DECLARATIONS All variable declarations in BX1 are prefixed with the keyword `var` and suffixed with the type of the variables being defined. Here are some examples:

```
var x : int64;              // uninitialized
var y = 20 : int64;         // initialized
var z = x + y : int64;      // can initialize with expressions
var u, v = 42, w = v + 20 : int64;
                            // can mix initialized and uninitialied vars;
                            // can also initialize wrt. other initialized vars
                            // declared earlier in the same "var" declaration

// var x, y, x : int64;     // illegal: x declared multiple times
// var m = m : int64;       // illegal: m unknown when trying to initialize m
// var m = n, n = 20 : int64;
                            // illegal: n unknown when trying to initialize m
var n = 20, m = n : int64;  // OK

var b1, b2 = true : bool;   // boolean vars
```

In the BX1 grammar, we add a new non-terminal called ⟨vardecl⟩ to stand for variable declarations. It is defined in terms of ⟨varinit⟩, which stands for a *possibly initialized variable*.

⟨vardecl⟩ ::= `"var"` ⟨varinit⟩ (`","` ⟨varinit⟩)* `":"` ⟨type⟩ `";"`
⟨varinit⟩ ::= ⟨variable⟩ (`"="` ⟨expr⟩)?

COMPARISON AND BOOLEAN OPERATORS BX1 adds the equality (==) and inequality (!=) operators between two expressions *of the same type*. Additionally, for integer expressions, BX1 adds the usual inequality relations (<, <=, >, >=). All such operators are binary and produce an expression of `bool` type.

BX1 also adds the *short-circuiting* version of boolean and (&&) and boolean or (||) along with boolean neagation (!). Short-circuiting means that when evaluating the arguments of the binary operators left-to-

right, if the left operand evaluates to a value that renders the right operand moot then the computation of the right operand is simply omitted. Specifically, in the following cases, the expression `E` is not computed.

```
u = false && E;
u = true || E;
```

Figure 1 contains the opdated operator precedence table.

BX1 Programs Unlike in BX0, in BX1 all valid ⟨program⟩s begin with a sequence of variable declarations, which are then followed by a sequence of statements. The statements are only allowed to use the declared variables. BX1 adds *three* new kinds of statements: ⟨block⟩s, ⟨ifelse⟩s, and ⟨while⟩s.

⟨program⟩ ::= ⟨vardecl⟩* ⟨statement⟩*

⟨statement⟩ ::= ⟨move⟩ | ⟨print⟩ | ⟨block⟩ | ⟨ifelse⟩ | ⟨while⟩

BX0 Statement Forms The ⟨move⟩ and ⟨print⟩ statement forms from BX0 continue to be valid BX1 statements, except now the terminal semi-colon is part of their syntax. In other words, semi-colons are not used to separate statements any more in BX1.

⟨move⟩ ::= ⟨variable⟩ `"="` ⟨expr⟩ `";"`

⟨print⟩ ::= `"print"` ⟨expression⟩ `";"`

Statement Blocks A sequence (possibly empty) of statements delimited with `{` and `}`.

⟨block⟩ ::= `"{"` ⟨statement⟩* `"}"`

Conditionals (`if ... else ...`) BX1 follows the form of C-like languages and adds a condition statement, ⟨ifelse⟩, that begins with the keyword `else`. Immediately after the parenthesized condition expression, there is a `block` that is executed if the condition computes to `true`. There is an optional `else` clause that handles the case where the condition is `false`: this clause can be a `block` that to be executed, or can be another ⟨ifelse⟩ statement to continue in an `if ... else if ...` chain.

⟨ifelse⟩ ::= `"if"` `"("` ⟨expr⟩ `")"` ⟨block⟩ (`"else"` (⟨ifelse⟩ | ⟨block⟩))?

Unstructured Loops (`while`) BX1 also uses the C-like `while` loops which repeatedly evaluate the condition expression until it becomes `false`. Every time the condition computes to `true`, the body of the loop (a ⟨block⟩) is executed.

⟨while⟩ ::= `"while"` `"("` ⟨expr⟩ `")"` ⟨block⟩

2.2 *Semantics and Type-Checking*

Not every BX1 program that can be parsed will be *well-formed*, and not all well-formed programs will *type-check*. From the parse tree that the parser builds based on the grammar, you will have to perform some *syntactic analysis* to build the abstract syntax tree (AST), and then perform *type-checking* to check that all the type restrictions are followed. Type-checking will also be used to annotate the internal nodes of the AST with the types of the corresponding subexpressions for use by later stages of the compiler.

Well-Formed BX1 Programs A BX1 program is considered to be well-formed if:

- Whenever a variable is used in the program, it occurs syntactically earlier in the variable declarations. Note that variables can also be used to initialize other variables.
- No *uninitialized* variable is read from before it is written to at least once.

The first restriction forces the programmer to make conscious decisions about all the variables they need to use in a BX1 program, while the second restriction catches inadvertent mistakes.

Type-Checking A well-formed BX1 program may nevertheless fail to have a sensible meaning because it does not respect the type discipline. In particular, BX1 has the following type restrictions:

- The arithmetic operators (`+`, `-`, `*`, `/`, `%`) and bitwise operators (`&`, `|`, `^`, `~`, `<<`, `>>`) are only allowed to have `int64` arguments and they build `int64` expressions.
- The inequality operators (`<`, `<=`, `>`, `>=`) must have `int64` arguments and they build `bool` expressions.
- The comparison operators (`==`, `!=`) must have two operands of the same type and they build `bool` expressions.
- The boolean operators (`&&`, `||`, `!`) must have `bool` arguments and they build `bool` expressions.

Note that the argument to `print` can be an expression of any type. As BX1 evolves we will try our best to retain this property of `print`: it will be able to display the result of any BX expression.

2.3 *Interpreter*

To help you with writing the compiler, we provide an *interpreter* for BX1 programs. The interpreter directly executes a program from its source code, without first compiling it to machine code. The BX1 interpreter is available at the following location in the lab computers:[2]

```
/users/profs/info/kaustuv.chaudhuri/CSE302/BX1/interpret.exe
```

To run `interpret.exe`, you have to give it the BX1 source file as its sole argument. For example:

```
$ /users/profs/info/kaustuv.chaudhuri/CSE302/BX1/interpret.exe fibs.bx
0
1
1
2
3
⋮
```

The interpreter will complain about ill-formed and ill-typed BX1 programs. You don't have to mimic the error messages of the interpreter verbatim. Just use it as a guide.

[2] I plan to put the interpreter online so you can run it in a browser. This document will be updated once I get this working.

3 WEEK 1 TASK: WRITING THE `BX1` PARSER

We will illustrate three styles of writing parsers: Antlr, Lex/Yacc, and roll-your-own recursive descent (not recommended). We will use the following simple EBNF grammar for a language named E for illustration.

⟨expr⟩ ::= ⟨number⟩ | ⟨expr⟩ `"+"` ⟨expr⟩ | ⟨expr⟩ `"*"` ⟨expr⟩ | `"("` ⟨expr⟩ `")"`
⟨number⟩ ::= `[0-9]+`

3.1 *Antlr Parsers*

The sample code that was given to you for lab 1 was based on a parser generator tool called Antlr4. Antlr4 takes a grammar description file that looks very similar to EBNF, and generates code for an efficient parser for that grammar. This generated parser internally builds a *parse tree* that closely mirrors the definition of the grammar. Antlr4 also generates code for a generic traversal class (called a *walker*) for the parse tree, which we then subclass to create any particular AST we desire.

To write an Antlr4-based parser, we have to first write a *grammar file*, which is a file with the `.g4` extension. Here is the above E grammar, written in `E.g4`.

```
grammar E ; // name of the grammar

// each production of a nonterminal is given a unique name
// written after the # character
expr: NUMBER               # number

    // the productions are listed in order of decreasing precedence
    // i.e., higher productions are reduced before lower ones
    | expr '*' expr        # times

    // individual subtrees can be given names with =
    | l=expr op='+' r=expr  # plus
    // here the first expr subtree is named l, the
    // operator text is named op, and the second expr
    // subtree is named r

    | '(' expr ')'         # parens
    // the list of productions always ends with a semi-colon
    ;

// capitalized names stand for terminals, and they are
// mapped to regular expressions
NUMBER: [0-9]+ ;

// there are special non-terminals for whitespace, comments, etc.
// that are ignored, indicated with skip
WHITESPACE: [ \t\r\n] -> skip ;
```

The syntax of such a grammar is fairly straightforward and easily translated from the EBNF. You may also use parentheses and the operators *, +, ?, etc. to indicate regular structures in the grammar.

To generate a grammar from such a grammar file, we have to run the Antlr4 tool on it. By default the tool will generate a parser written in Java, but you can change the language to C++, Python, etc. using the `-Dlanguage=` option. The output directory can be specified with the `-o` option.

```
$ java -jar .../antlr-4.7.2-complete.jar -o java_parser E.g4

$ ls java_parser
E.interp       E.tokens        EBaseListener.java  ELexer.interp  ELexer.java
ELexer.tokens  EListener.java  EParser.java

$ java -jar .../antlr-4.7.2-complete.jar -o cpp_parser -Dlanguage=Cpp E.g4

$ ls cpp_parser
E.interp  EBaseListener.cpp  ELexer.cpp  ELexer.interp  EListener.cpp  EParser.cpp
E.tokens  EBaseListener.h    ELexer.h    ELexer.tokens  EListener.h    EParser.h
```

From Files to Parse Trees The code that Antlr4 generates can be used to transform a syntactically valid E source file into a *parse tree*, which are subclasses of the class `ParseTree` (found in the Antlr4 library, called its *runtime*). Here is how to generate such a parse tree in Java:

```
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;

// ExprContext is a subclass of ParseTree corresponding
// to a subtree holding with an expr at the root node
class E {
  public static EParser.ExprContext parseFile(String file) throws Exception {
     CharStream input = CharStreams.fromFileName(file);
     // the lexer class is named <grammarname>Lexer
     ELexer lexer = new ELexer(input);
     // use the lexer to make a stream of tokens
     CommonTokenStream tokens = new CommonTokenStream(lexer);
     // create a parser that reads this token stream;
     // the parser class is named <grammarname>Parser
     EParser parser = new EParser(tokens);
     // every nonterminal is a function in the parser object
     return parser.expr();
  }
}
```

And here it is in C++:

```
#include <iostream>
#include "ELexer.h"
#include "EParser.h"

void process_e(std::string file) {
  std::ifstream stream;
  stream.open(file);
  antlr4::ANTLRInputStream input{stream};
  ELexer lexer{&input};
  antlr4::CommonTokenStream tokens{&lexer};
  EParser parser{&tokens};
  antlr4::tree::ParseTree *tree = parser->expr();
  // now we process the tree
}
```

Walking a Parse Tree The `ParseTree` object that is returned is a fairly generic tree implementation that you can modify with your own functions. However, one standard style of using the parse tree is to perform an inorder traversal of the parse tree and use that traversal to compute an AST. This is where the `Listener` class comes in: it is just a class with a collection of `enter`/`exit` callbacks, one per name of a production (i.e., the stuff after # in `E.g4`). This is what it looks like for the Java case.

```
public interface EListener
  extends org.antlr.v4.runtime.tree.ParseTreeListener
{
  void enterNumber(EParser.NumberContext ctx);
  void exitNumber(EParser.NumberContext ctx);
  void enterParens(EParser.ParensContext ctx);
  void exitParens(EParser.ParensContext ctx);
  void enterTimes(EParser.TimesContext ctx);
  void exitTimes(EParser.TimesContext ctx);
  void enterPlus(EParser.PlusContext ctx);
  void exitPlus(EParser.PlusContext ctx);
}
```

As the walker object is walking the parse tree, whenever it starts to visit an internal node that has been produced by a production named `N`, it calls the `enterN()` function. Likewise, after it has fully walked the subtree rooted at this node, it calls the `exitN()` function. In both cases, the argument to the function for a production named `N` is the corresponding instance of a `ParseTree` subclass that holds the node, which is named like `EParser.NContext`.

It is your job to write a subclass of this `EListener` class by filling in the details of all the functions. In general, the `exit()` functions are what you will want to override and use them to build the AST subtree corresponding to a given production. For example, suppose we define the AST for expressions as follows:

```
class Expr { }
class Number extends Expr {
   public final int value;
   public Number(int value) { this.value = value; }
}
class Binop extends Expr {
   public final Expr left, right;
   public static enum Op { PLUS, TIMES; }
   public final Op op;
   public Binop(Expr left, Op op, Expr right) {
     this.left = left;
     this.op = op;
     this.right = right;
   }
}
```

Our `EListener` subclass needs to build the corresponding `Expr` subclass as it traverses the parse tree (subclass) returned by `E.parseFile()`. Since the `exit()` methods do not return anything, we will have to explicitly manage the "call stack" ourselves by maintaining a stack of `Expr` instances (corresponding to the siblings to the left of the parse tree). Here is how it looks.

```
// EBaseListener is an implementation of EListener that does
// nothing for all the methods; we will override the exit() methods
class EReader extends EBaseListener {
  public List<Expr> stack = new ArrayList<>();

  // all terminals just get pushed on the stack
  @Override void exitNumber(EParser.NumberContext ctx) {
    String txt = ctx.getText(); // the literal text that was parsed
    this.stack.push(new Number(Integer.parseInt(txt)));
  }

  // for non-terminals, the top of the stack contains the
  // operands, so we pop them off, build the AST node, and then
  // push the node back onto the stack

  @Override void exitTimes(EParser.TimesContext ctx) {
     // the operands are on the stack in reverse order
     Expr right = this.stack.pop();
     Expr left = this.stack.pop();
     // as a sanity, check that the operator was correct
     assert(ctx.op().getText().equals("*"));
     this.stack.push(new Binop(left, Binop.Op.TIMES, right));
  }

  @Override void exitPlus(EParser.PlusContext ctx) {
     Expr right = this.stack.pop();
     Expr left = this.stack.pop();
     // we *know* that the operator was "+", so no need to check
     this.stack.push(new Binop(left, Binop.Op.PLUS, right));
  }
}
```

Using this subclass of `EListener` is fairly simple: we create a walker object that traveres the parse tree using our listener, and in the end we retrieve the top of its stack (which would contain the contents of the root-most `exit()` call). Let us modify our `parseFile()` method.

```
class E {
  public static Expr parseFile(String file) throws Exception {
     // ... the stuff from before up to:
     EParser parser = new EParser(tokens);
     EReader reader = new EReader(); // our listener
     ParseTreeWalker walker = new ParseTreeWalker(); // the inorder traverser
     walker.walk(reader, parser.expr());
     // now reader's stack contains the final Expr that was built
     return reader.stack.pop();
  }
}
```

The C++ version of this code is pretty similar. You can use the above guide together with the sample solution to lab 1 to see how to do it in C++.

3.2 *Lex/Yacc Parsers*

A more traditional alternative to the Antlr4 style is the Lex/Yacc style that consists of:

- A lexical scanner (lex), which transforms input text to tokens based on a lexer specification. Modern implementations are `flex` (C/C++), JFlex (Java), `ocamllex` (OCaml), etc.
- A parser generator (Yacc), which builds efficient parsers of tokens based on a grammar slecification. These parser generators usually limit themselves to a fragment of LR grammars, generally LALR(k) for $k \leq 2$, for reasons of efficiency. Modern implementations are `bison` (C/C++), Java-CUP (Java), Menhir (OCaml), etc.

Here we will give an example of this style of parsing in OCaml, using `ocamllex` and Menhir. Our target will be the following definition of the AST for the E language.

```
type expr = Num   of int
          | Plus  of expr * expr
          | Times of expr * expr ;;
```

The Grammar Specification In the Lex/Yacc style, it is common to write the grammar specification before writing the lexer specification, since the former informs what tokens the latter needs to scan for. The grammar file begins with a declaration of the terminal tokens, followed by a definition of the precedence and associativity of the operators. Then, the rules of the grammar are written with each rule annotated with a code snippet that assembles a value from the components of the parsed rule. Here is how it looks.

```
%token <int> NUM   /* NUM terminals have an associated int value */
/* other tokens have no values */
%token PLUS TIMES LPAREN RPAREN
%token ENDOFSTREAM

/* operators and arities ordered by increasing precedence */
%left PLUS
%left TIMES

/* the start symbol */
%start <expr> expr

/* the nonterminal section is separated from the above by %% */
%%

expr:
  | LPAREN e=expr RPAREN     /* this is the production */
    { e }                    /* this is the associated return value */
  | n=NUM
    { Num n }
  | l=expr PLUS r=expr       /* parts of the production are named using = */
    { Plus(l, r) }
  | l=expr TIMES r=expr
    { Times(l, r) }          /* Inside the { } is just normal OCaml code */
```

This grammar specification is placed in a file ending in `.mly`, such as `eParser.mly`, which is then fed to the Menhir program to generate the corresponding parser in `eParser.ml`.

```
$ menhir --strict --infer eParser.mly
```

The Lexer Specification The tokens used by the grammar are conveniently listed in `eParser.mly`, so the lexer specification is just a mapping from regular expressions to these tokens. The `EParser.token` type is an abstract data type where each token name is a constructor and the token values (if any) are the arguments to the constructor. For example, for teh above grammar we would have a generated datatype (in `eParser.ml`) such as:

```
type token = NUM of int
           | PLUS | TIMES | LPAREN | RPAREN
           | ENDOFSTREAM ;;
```

The lexer specification will define a sequence of mappings from regular expressions to these tokens. These rules are put in a file such as `eLexer.mll` with the following contents.

```
(* some regular expressions can be named for convenience *)
let number = ['0'-'9']+

let whitespace = ' ' | '\t' | '\r'

(* the actual tokenizer is named "token", but this name is arbitrary *)
rule token = parse
| whitespace  { token lexbuf (* just run the token rule again *) }
| number as n { EParser.NUM (int_of_string n) }
| '('         { EParser.LPAREN }
| ')'         { EParser.RPAREN }
| '+'         { EParser.PLUS }
| '*'         { EParser.TIMES }
(* eof is a special name for the end of the input *)
| eof         { EParser.ENDOFSTREAM }
(* newlines are used to increment a line count *)
| '\n'        { Lexing.new_line lexbuf ; token lexbuf }
(* any other character is an error *)
| _ as c      { failwith ("Unexpected character: " ^ c) }
```

To generate the lexical scanner, we use `ocamllex`, which produces `eLexer.ml`.

```
$ ocamllex eLexer.mll
```

Assembling the Parser The modules `ELexer` and `EParser` can now be used to write a function that converts a file to an `expr`:

```
let parse_file filename =
  let chan = open_in filename in
  at_exit (fun () -> close_in chan) ;  (* could also close after parsing *)
  let lbuf = Lexing.from_channel chan in
  (* (Optional) inform the lexer which "file" it is coming from *)
  lbuf.lex_curr_p <- {lbuf.lex_curr_p with pos_fname = filename} ;
  EParser.prog ELexer.token lbuf ;;
```

3.3 *Roll-Your-Own*

If you want ultimate control over the code of your parser, you will probably want to write the parser by hand, generally following a *recursive descent* style. This trades of speed and ease of maintenance for

complete flexibility and the ability to create finely tuned error messages.

The Wikipedia article on recursive descent parsing does an admirable job of giving an overview of this area, together with examples in a number of programming languages. It is hard to do much better than this in terms of general guidance. If you're rolling your own parser and are facing difficulties, please contact the instructor ASAP.

4 WEEK 2 TASK: COMPILING `BX1`

It is recommended that you compile `BX1` into assembly in two steps: first compiling it to an intermediate language called *Register Transfer Language* (RTL), and then transforming the RTL to `AMD64` assembly.

4.1 *Register Transfer Language (RTL)*

The purpose of RTL is to remove all the structured control in the `BX1` program and replace it with *jumps*. To this end, the basic concept introduced in RTL is that of a *label*. Broadly speaking, a label is where an instruction can be stored. We will write labels using the syntax `L0`, `L1`, etc.

RTL also contains a number of *instructions*. Every instruction can refer to one or several operands, which can be immediate values or the names of *pseudo registers*. We will write pseudo registers as `#1`, `#2`, etc. Each pseudo register has a fixed type (`int64` or `bool`) that never changes, and there is an infinite supply of pseudo registers. The special pseudo register `##` stands for *don't care*: its value will never be read, and its sole purpose is to stand in for a destination where one is syntactically needed but otherwise irrelevant.

Every RTL instruction can also optionally have zero, one, or two *out labels*. These are the labels of the instructions to run after the current instruction. Only one instruction has no out labels, the `return` instruction, which is intended to end the program. If an instruction has only one out label, that's the next instruction to run. If it has two out labels, then the instruction encodes a test, and the first label is the case where the test succeeded, while the second label denotes the path taken when the test fails.

In addition to the labels for instructions, there are two labels for the entire `BX1` program: *enter*, and *leave*. The *enter* label is the label of the first instruction to be executed, while the *leave* label is the label of the final reachable instruction of program, which is by convention going to be the `return` instruction.

The full set of RTL instructions together is given below. Each instruction is accompanied by some concrete syntax that can be used to print the instructions. In fact, the entire RTL program can be printed to a file and then simulated using an RTL interpreter we provide.

```
$ /users/profs/info/kaustuv.chaudhuri/CSE302/BX1/rtl_interpret.exe fibs.rtl
0
1
1
2
3
⋮
```

4.1.1 *Sequential Instructions (Single Out-Labels)*

- `move`: this instruction is used to store an immediate in a pseudo register.

  ⟨rtl-move⟩ ::= `"move"` ⟨number⟩ `","` ⟨pseudo⟩ `"-->"` ⟨label⟩

- `copy`: this instruction copies the value of the first pseudo register to the second.

  ⟨rtl-copy⟩ ::= `"copy"` ⟨pseudo⟩ `","` ⟨pseudo⟩ `"-->"` ⟨label⟩

- `unop`: unary operations on a single `int64` pseudo register. These instructions are differentiated by a special ⟨unop-code⟩.

  ⟨rtl-unop⟩ ::= `"unop"` ⟨unop-code⟩ `","` ⟨pseudo⟩ `"-->"` ⟨label⟩

⟨unop-code⟩	meaning
`"neg"`	`arg = - arg`
`"not"`	`arg = ~ arg`

- `binop`: binary operations on two `int64` pseudo registers. The second pseudo register (the `dest`) will hold the result of combinging its current value with that of the first pseudo register (the `src`).

  ⟨rtl-binop⟩ ::= `"binop"` ⟨binop-code⟩ `","` ⟨pseudo⟩ `","` ⟨pseudo⟩ `"-->"` ⟨label⟩

⟨binop-code⟩	meaning
`"add"`	`dest = dest + src`
`"sub"`	`dest = dest - src`
`"mul"`	`dest = dest * src`
`"div"`	`dest = dest / src`
`"mod"`	`dest = dest % src`
`"sal"`	`dest = dest << src`
`"sar"`	`dest = dest >> src`
`"and"`	`dest = dest & src`
`"or"`	`dest = dest | src`
`"xor"`	`dest = dest ^ src`

4.1.2 *Branches (Two Out-Labels)*

Each of these instructions have two out-labels, which we write as `Lt` and `Lf`, standing for a successful and unsuccessful test respectively.

- `ubranch`: this tests a single `int64` pseudo register.

  ⟨rtl-ubranch⟩ ::= `"ubranch"` ⟨ubranch-code⟩ `","` ⟨pseudo⟩ `"-->"` ⟨label⟩, ⟨label⟩

⟨ubranch-code⟩	jump to `Lt` if
`"jz"`	`arg == 0`
`"jnz"`	`arg != 0`

- `ubranch`: this tests a pair of `int64` pseudo registers.

⟨rtl-bbranch⟩ ::= "bbranch" ⟨bbranch-code⟩ "," ⟨pseudo⟩ "," ⟨pseudo⟩
"-->" ⟨label⟩, ⟨label⟩

⟨bbranch-code⟩	jump to `Lt` if
"je"	`arg1 == arg2`
"jne"	`arg1 != arg2`
"jg", "jnle"	`arg1 > arg2`
"jge", "jng"	`arg1 >= arg2`
"jl", "jnge"	`arg1 < arg2`
"jle", "jng"	`arg1 <= arg2`

4.1.3 *Other Instructions*

- `goto`: an unconditional jump to an out-label. Does not have any operands.

  ⟨rtl-goto⟩ ::= "goto" "-->" ⟨label⟩

- `call`: function calls take a function name and a list of arguments (all `int64` pseudo registers) separated by commas, and a destination pseudo register.

  ⟨rtl-call⟩ ::= "call" ⟨variable⟩ "(" (⟨pseudo⟩ ("," ⟨pseudo⟩)*)? ")" "," ⟨pseudo⟩
  "-->" ⟨label⟩

- `return`: a `BX1` program terminates with an exit code, an `int64` integer, specified as the argument to the `return` instruction. Note that this instruction does not have an out-label, since once it is executed the `BX1` program halts.

  ⟨rtl-return⟩ ::= "return" ⟨pseudo⟩

4.1.4 *The RTL File (Full Grammar)*

An RTL program is given by the nonterminal ⟨rtl-program⟩ that consists of the following productions together with the rest of the productions in this section.

⟨rtl-program⟩ ::=
"enter:" ⟨label⟩ "\n"
"leave:" ⟨label⟩ "\n"
"----\n"
⟨rtl-labeled-instr⟩*

⟨rtl-labeled-instr⟩ ::= ⟨label⟩ ":" ⟨rtl-instr⟩ "\n"

⟨rtl-instr⟩ ::= ⟨rtl-move⟩ | ⟨rtl-copy⟩ | ⟨rtl-unop⟩ | ⟨rtl-binop⟩
| ⟨rtl-ubranch⟩ | ⟨rtl-bbranch⟩ | ⟨rtl-goto⟩ | ⟨rtl-call⟩ | ⟨rtl-return⟩

4.2 *Generating and Compiling RTL*

RTL generation from BX1 source will require using one of the two maximal munch techniques. The top-down maximal munch technique was explained in the lecture slides. Its bottom-up variant can be reconstructed from this description and your intuitions from labs 1 and 2.

Compiling RTL to AMD64 is fairly straightforward. Each instruction is prefixed with its label in assembly code. Note that assembly labels that begin with `.` are considered to be local to the compilation unit (file), i.e., they are not exported as symbols in the object code. ALso note that all labels, global or local, are required to be unique in the entire compilation unit.

We recommend that you translate an RTL label such as `L42` into a local AMD64 label such as `.Lmain.42`. This will become useful in future labs when we add other functions and procedures besides `main()`.

As an optimization of the generated AMD64 assembly, you may remove some redundant jumps in the following situations.

- Jumps to immediate next label, which is a situation such as the following.

```
1       # ...
2       jmp .Lmain.42
3
4   .Lmain.42:
5       # ...
```

  In this case, the `jmp` on line 2 can be simply deleted. Note that the label `.Lmain.42` on line 4 cannot be deleted, since other code may still need to jump there.

- Labels for unconditional jumps, which is the following situation:

```
1       jmp .Lmain.42:
2
3       # ...
4
5   .Lmain.42:
6       jmp .lmain.64
7
8       # ...
9
10  .Lmain.64:
11      # ...
```

  In this case the intermediate jump location `.Lmain.42` can be eliminated by globally replacing all instances of `.Lmain.42` with `.Lmain.64`. The `jmp` in line 6 should remain untouched because code from before `.Lmain.42` might still lead into it.