The course project is to build a compiler for a small language. This is a "living" document will be revised throughout the semester until it is a complete, if sometimes informal, language specification. Revisions may include additions, removals and changes to meet pedagogical goals and to ensure internal consistency.

(X) means zero or one occurrence of X { X } + means one or more occurrences of X { X } * means zero or more occurrences of X

SECTION 1: Lexical structure (see version 1.1 of this document)

SECTION 2: Syntactic structure

The language is defined (informally) as follows; part of your job is to define a reasonable formal grammar that you can use with lex and yacc to parse/compile programs written in the language. **Keywords** appear in bold.

program is:

definition-list sblock

definition-list is:

definition definition-list | e

definition is:

Only permitted at top-level.
Defines a new type or function

type identifier ':' dblock

Record type.

type identifier$_1$ ':' constant '->' identifier$_2$ (':' '(' constant ')')

Array type.
identifier1 is name of arraytype.
constant: an integer, the number of dimensions.
identifier2 is name of element type.
constant: optional initial value for all elements.

type identifier ':' pblock '->' identifier

Function type.

function identifier$_1$ ':' identifier$_2$ sblock

Function definition.
identifier1 is function name,
identifier2 is function type)

sblock is:

'{' (dblock) statement-list '}'

sblock allows local declarations in optional dblock.

dblock is:

'[' declaration-list ']'

declaration-list is:

declaration ';' declaration-list | declaration

declaration is:

identifier ':' identifier-list *LHS is type, RHS is list of variable names.*

identifier-list is:

identifier (assignOp constant) ',' identifier-list | identifier (assignOp constant)

statement-list is:

statement statement-list | statement

statement is: *Recall: a statement has no value.*

for '(' statement ';' expression ';' statement ')' sblock *Boolean expression*

while '(' expression ')' sblock *Boolean expression*

if '(' expression ')' **then** sblock **else** sblock *Boolean expression, else is required*

switch '(' expression ')' { **case** constant ':' sblock }+ **otherwise** ':' sblock
integer expression
Implicit break at sblock end

sblock *Nested block & therefore nested scope*

assignable assignOp expression ';'

memOp assignable ';'

assignable is:

identifier — *Variable*

assignable ablock — *Function call or array access. Can be assigned to only as an array access. Size of ablock must match number of array dimensions (for array access) or number of parameters (for function call). For array access each member of ablock must be an integer, denoting the size of the corresponding dimension. For function call each member of ablock must be of the correct type, as determined by function's type. We follow Pascal convention of indicating return value by assigning the return value to an implicitly declared variable whose name is the name of the function. No coercion. Pass-by-value, as in Java.*

assignable recOp identifier — *Record access*

expression is: — *Recall: an expression has a value.*

constant — *Literal, e.g. 17, **`true`**, **`false`**, **`null`**, "foo".*

preUnaryOperator expression

expression postUnaryOperator

assignable

expression binaryOperator expression

'(' expression ')' — *Parenthesized expression.*

pblock is:

'(' parameter-list ')' — *pblock must have parentheses.*

parameter-list is:

non-empty-parameter-list | e — *Parameter list can be empty.*

non-empty-parameter-list is:

parameter-declaration ',' non-empty-parameter-list | parameter-declaration

parameter-declaration is:

identifier ':' identifier	*LHS is type, RHS is variable name.*

ablock is:

'(' argument-list ')'	*ablock must have parentheses.*

argument-list is:

non-empty-argument-list | e	*Argument list can be empty.*

non-empty-argument-list is:

expression ',' non-empty-argument-list | expression

preUnaryOperator is:

-	*Numeric negation.*
!	*Logical negation.*
i2r	*integer to real type conversion.*
r2i	*real to integer type conversion.*

postUnaryOperator is:

isNull	*Returns true for null, false otherwise.*

memOp is:

reserve	*Allocates space for type object.*
release	*Releases space for type object.*

assignOp is:

:=	*Assignment, no coercion.*

recOp is:

.	*Record access.*

binaryOperator is:	*Usual prec/assoc rules apply.*
+, -, *, / (both integer and real)	*Coercion from integer->real only.*
% (integer only)	*No coercion.*
&, |	*Logical operators, short circuiting.*
<, =	*Relational operators.* *Coercion from integer->real only.*
defined for numeric types:	*i*i->b, r*r->b, c*c->b (numeric '<')*
defined for Boolean:	*b*b>b (false<true)*

Functionality from previous stages

Your submission should not only provide the functionality of this stage, but also that described in previous project stages.

Invoking your (partial) compiler with input from a file will lex the file and hand off tokens to the syntactic parser. The syntactic parser will build various internal data structures, whose details are left to you and your teammates.

With no options the compiler must write error messages to standard output.

Given relevant options the compiler must write the source program, annotated with line numbers, error messages and scope indications, to a file.

Given relevant options the compiler must write the symbol table to a file.

Ex: Assume **comp** is the name of your compiler. Assume **prog** contains

```
(* Type definition *)
type unaryIntFunction: (integer: x) -> integer


(* This is a function definition.
   It uses the above type definition.
*)
function square : unaryIntFunction {
```

```
    square := x * x;
}

(* This is the main block of the program.
   Execution begins in this block.
*)
{
  [ integer: input := 7, expected := 49, actual ; boolean: result ]
  actual := square(input);
  rseult := expected = actual;
}
```

Invoking

comp prog

should lex and parse the contents of **prog**. Any error messages must be printed to the console, as in:

```
LINE 15:51 ** ERROR: the name 'boolean', used here as a type, has not been declared at this p
oint in the program.
LINE 17:3 ** ERROR: the name 'rseult', used here as a variable name, has not been declared at
this point in the program.
```

Error messages should all begin with "LINE lineNumber:columnNumber ** ERROR:", and then give a description of what the error was.

Invoking

comp -asc prog

should lex and parse the contents of **prog** and produce **a**nnotated **s**ource **c**ode to the file prog.asc

In this case the source code listing contained in prog.asc should be:

```
01:0:   (* Type definition *)
02:0:1: type unaryIntFunction: (integer: x) -> integer
03:0:
04:0:   (* This is a function definition.
  It uses the above type definition.
*)
07:0:2: function square : unaryIntFunction {
08:0:2:    square := x * x;
```

```
09:0:2: }
10:0:
11:0:   (* This is the main block of the program.
   Execution begins in this block.
*)
14:0:3  {
15:0:3    [ integer: input := 7, expected := 49, actual ; boolean: result ]
** ERROR:15:51: the name 'boolean', used here as a type, has not been declared at this point
in the program.
16:0:3    actual := square(input);
17:0:3    rseult := expected = actual;
** ERROR:17:3: the name 'rseult', used here as a variable name, has not been declared at this
point in the program.
18:0:3  }
```

Error messages should all begin with "** ERROR:lineNumber:columnNumber:", and have a description of the error that was found. The description of the error need not be exactly as shown (you should come up with messages that are as meaningful as you can make them, without being overly wordy). Your parser must produce error messages for errors identified by the LALR parse table, as well as undeclared names identified by symbol table lookup. For this example there may be other errors that your parser identifies, in which case they should be included in the parser output as well. The above is not intended to be a definite statement of the parser's output, but an indication of the format expected.

Invoking

comp -st prog

must lex and parse the contents of **prog** and produce a **s**ymbol **t**able description to the file prog.st

The symbol table must be written to the file as follows:

```
NAME             : SCOPE : TYPE                : Extra annotation
unaryIntFunction : 0     : (integer) -> real   : type
x                : 1     : integer             : parameter
square           : 0     : unaryIntFunction    : function
input            : 3     : integer             : local
expected         : 3     : integer             : local
actual           : 3     : integer             : local
result           : 3     : boolean             : local
```

SUBMISSION & GRADING:

Submit your code using AutoLab. Submissions are due no later than 5:00 PM on Monday March 12. Please be aware that Prof. Alphonce will be out of town Feb 22 (no office hours this day) through Feb 24, and e-mail response during this time will be slower than usual.