CS 475/575 **Programming Project: ToyL** **Spring 2020**

ToyL is short for "Toy Language". First we quickly summarize the lexical conventions of ToyL, then we discuss the syntax and semantics of ToyL in detail.

Comments are enclosed between /∗ and ∗/. Other white space includes blanks, newlines, and tabs. Case (upper or lower) is not significant. An *identifier* begins with a letter, and may include letters, digits, and underscores. A *number* consists only of one or more digits. The reserved words, operation symbols, and punctuation marks can be extracted from the syntax below.

A grammar for ToyL is described below using a variant of Backus Normal Form (BNF). Each production matches this template.

$\langle$*nonterminal-symbol*$\rangle \longrightarrow$ *definition*

Several typographical conventions will be observed.

$\langle$*foo*$\rangle$	*foo* is a nonterminal symbol.
foo	*foo* is a terminal symbol.
$\boxed{\texttt{foo}}$	foo is a terminal symbol, represented as itself.
[foo]	the symbol or expression foo is optional, occuring zero or one times.
{foo}	the symbol or expression foo may be repeated any number of times.
(a | b | c)	exactly one of a or b or c must appear.

The grammar for ToyL now follows.

Syntax and Semantics of ToyL

The main program is merely the outermost procedure, but it must not have any formal parameters. Each procedure name is repeated at the end of the procedure, and these two identifiers must match.

$\langle$*program*$\rangle \longrightarrow \langle$*procedure*$\rangle$

$\langle$*procedure*$\rangle \longrightarrow$ $\boxed{\texttt{procedure}}$ *identifier* [$\langle$*formals*$\rangle$] $\boxed{\texttt{is}}$
$\langle$*declarations*$\rangle$
$\boxed{\texttt{begin}}$
$\langle$*stmtlist*$\rangle$
$\boxed{\texttt{end}}$ *identifier* $\boxed{\texttt{;}}$

A procedure's heading can be given without providing its body, but in this case a complete declaration with a body must be provided later. This is similar to "forward" or incomplete declarations in other languages. It is useful for writing mutually recursive procedures, and also for exporting interfaces from packages.

$\langle$*procedure*$\rangle \longrightarrow$ $\boxed{\texttt{procedure}}$ *identifier* [$\langle$*formals*$\rangle$] $\boxed{\texttt{;}}$

Functions are declared almost like procedures, except that they are introduced with the keyword $\boxed{\texttt{function}}$, and they have return values.

$$\begin{array}{lll}
\langle \textit{function} \rangle & \longrightarrow & \boxed{\texttt{function}}\ \textit{identifier}\ [\,\langle \textit{formals} \rangle\,]\ \boxed{\texttt{return}}\ \langle \textit{type-identifier} \rangle\ \boxed{\texttt{is}} \\
 & & \langle \textit{declarations} \rangle \\
 & & \boxed{\texttt{begin}} \\
 & & \langle \textit{stmtlist} \rangle \\
 & & \boxed{\texttt{end}}\ \textit{identifier}\ \boxed{\texttt{;}} \\
\langle \textit{function} \rangle & \longrightarrow & \boxed{\texttt{function}}\ \textit{identifier}\ [\,\langle \textit{formals} \rangle\,]\ \boxed{\texttt{return}}\ \langle \textit{type-identifier} \rangle\ \boxed{\texttt{;}} \\
\langle \textit{formals} \rangle & \longrightarrow & \boxed{\texttt{(}}\ \{\langle \textit{formal} \rangle\ \boxed{\texttt{;}}\ \}\ [\,\langle \textit{formal} \rangle\ \,]\ \boxed{\texttt{)}} \\
\langle \textit{formal} \rangle & \longrightarrow & \{\ \textit{identifier}\ \boxed{\texttt{,}}\ \}\ \textit{identifier}\ \boxed{\texttt{:}}\ [\,\langle \textit{mode} \rangle\,]\ \langle \textit{type-identifier} \rangle \\
\langle \textit{mode} \rangle & \longrightarrow & [\ (\ \boxed{\texttt{in}}\ \mid \boxed{\texttt{out}}\ \mid \boxed{\texttt{in}}\ \boxed{\texttt{out}}\ )\,]
\end{array}$$

Parameter modes "`in`," "`out`," and "`in out`" refer to parameters passed by value, by result, and by reference, respectively. If the parameter mode is omitted, mode `in` is the default. Functions must have only `in` parameters.

$$\langle \textit{declarations} \rangle \longrightarrow \{\ \langle \textit{typedecl} \rangle\ \mid \langle \textit{vardecl} \rangle\ \mid \langle \textit{procedure} \rangle\ \mid \langle \textit{function} \rangle\ \mid \langle \textit{package} \rangle\ \}$$

A set of type, variable, procedure, function, and package declarations may be placed between the procedure header and the body. Visibility rules are based on lexical scope.

$$\begin{array}{lll}
\langle \textit{package} \rangle & \longrightarrow & \boxed{\texttt{package}}\ \textit{identifier}\ \boxed{\texttt{is}} \\
 & & \langle \textit{declarations} \rangle \\
 & & \boxed{\texttt{private}} \\
 & & \langle \textit{declarations} \rangle \\
 & & \boxed{\texttt{end}}\ \textit{identifier}\ \boxed{\texttt{;}}
\end{array}$$

A package is a scoping mechanism. The first set of declarations are called the "public" part of the package. Identifiers declared within the public part are exported, but they must be qualified with the package name. (For example, we write `packagename.x` to refer to a variable `x` in the public part of a package, and `packagename.foo(y)` to call a public procedure or function.)

The second set of declarations (after $\boxed{\texttt{private}}$) are invisible outside the package. It is usual to declare incomplete types and forward procedures in the public part, and to provide details of data structures and procedures in the private part, although there are no rules enforcing this policy. Note that if an incomplete type is declared in the public part, and then completed in the private part, the details of the type are inaccessible outside the package.

$$\begin{array}{lll}
\langle \textit{typedecl} \rangle & \longrightarrow & \boxed{\texttt{type}}\ \textit{identifier}\ \boxed{\texttt{is}}\ \langle \textit{typedesc} \rangle\ \boxed{\texttt{;}} \\
\langle \textit{typedecl} \rangle & \longrightarrow & \boxed{\texttt{type}}\ \textit{identifier}\ \boxed{\texttt{;}}
\end{array}$$

Some restrictions apply to incomplete types. In particular, components of composite types (records and arrays) must not be incomplete. This implies that the size of a record or array element is always known when the record or array type is declared.

The simplest type description is the name of an existing type. ⟨*type-identifier*⟩ is grammatically identical to an ⟨*l-expression*⟩, but it must be the name of a type that has been previously declared and is visible in the current scope.

$\langle \textit{typedesc} \rangle \longrightarrow \langle \textit{type-identifier} \rangle$

$\langle \textit{type-identifier} \rangle \longrightarrow \langle \textit{l-expression} \rangle$

$\langle \textit{type-identifier} \rangle \longrightarrow \boxed{\texttt{integer}} \mid \boxed{\texttt{boolean}}$

The built-in scalar types are `integer` and `boolean`. The type constructors are `record` and `array.`

$\langle \textit{typedesc} \rangle \longrightarrow \boxed{\texttt{array}}\ \boxed{\texttt{[}}\ \langle \textit{r-expression} \rangle\ \boxed{\texttt{..}}\ \langle \textit{r-expression} \rangle\ \boxed{\texttt{]}}\ \boxed{\texttt{of}}\ \langle \textit{typedesc} \rangle$

The elements of an array may be of any type. Array indices must be integers, and the bounds of an array are required to be *static* integer expressions, i.e., it must be possible to evaluate them to integers at compile time.

$\langle \textit{typedesc} \rangle \longrightarrow \boxed{\texttt{record}}$
$\quad \{\ \{\textit{identifier}\ \boxed{,}\}\ \textit{identifier}\ \boxed{:}\ \langle \textit{typedesc} \rangle\ \boxed{;}\ \}$
$\quad \boxed{\texttt{end}}\ \boxed{\texttt{record}}$

Note that every field, including the last, is terminated with a semicolon.

$\langle \textit{vardecl} \rangle \longrightarrow \{\ \textit{identifier}\ \boxed{,}\}\ \textit{identifier}\ \boxed{:}\ \langle \textit{typedesc} \rangle\ \boxed{;}$

An incomplete type name can not appear in a variable declaration. The type equivalence rule in ToyL is name equivalence.

The body of a procedure, including the main program, is a sequence of statements.

$\langle \textit{stmtlist} \rangle \longrightarrow \langle \textit{statement} \rangle\ \{\langle \textit{statement} \rangle\}$

Note that a statement list must contain at least one statement. Rather than creating an empty list of statements, one can use the "do nothing" statement called "`null`."

$\langle \textit{statement} \rangle \longrightarrow \boxed{\texttt{null}}\ \boxed{;}$

Functions should contain "`return`" statements (as in C) indicating the values to be returned. In a procedure, the expression is omitted from the return statement.

$\langle\textit{statement}\rangle \longrightarrow \boxed{\texttt{return}}\ [\ \langle\textit{r-expression}\rangle\ ]\ \boxed{\texttt{;}}$

Conditionals and looping statements are similar to languages such as Algol/Pascal/Ada.

$$\begin{array}{lll}
\langle\textit{statement}\rangle & \longrightarrow & \boxed{\texttt{if}}\ \langle\textit{r-expression}\rangle\ \boxed{\texttt{then}}\ \langle\textit{stmtlist}\rangle \\
 & & \{\boxed{\texttt{elsif}}\ \langle\textit{r-expression}\rangle\ \boxed{\texttt{then}}\langle\textit{stmtlist}\rangle\} \\
 & & [\boxed{\texttt{else}}\ \langle\textit{stmtlist}\rangle\ ] \\
 & & \boxed{\texttt{end}}\ \boxed{\texttt{if}}\ \boxed{\texttt{;}} \\
\langle\textit{statement}\rangle & \longrightarrow & \boxed{\texttt{while}}\ \langle\textit{r-expression}\rangle\ \boxed{\texttt{loop}} \\
 & & \langle\textit{stmtlist}\rangle\ \boxed{\texttt{end}}\ \boxed{\texttt{loop}}\ \boxed{\texttt{;}}
\end{array}$$

Another kind of statement is the procedure call.

$$\langle\textit{statement}\rangle \longrightarrow \langle\textit{proc-name}\rangle\ \boxed{\texttt{(}}\ [\{\ \langle\textit{r-expression}\rangle\ \boxed{\texttt{,}}\}\ \langle\textit{r-expression}\rangle\ ]\ \boxed{\texttt{)}}\ \boxed{\texttt{;}}$$

The number and types of arguments must match the procedure declaration, of course. Also, if the mode of an argument is "`out`" or "`in out`," then the actual argument must be an ⟨*l-expression*⟩, which will be defined below.

Procedure names can be simple identifiers, but procedures in packages are named with "dot" notation. For instance, `foo.bar(i)` is a call on procedure `bar` in the visible part of package `foo`. We therefore recognize it as an ⟨*l-expression*⟩.

$$\langle\textit{proc-name}\rangle \longrightarrow \langle\textit{l-expression}\rangle$$

$$\langle\textit{func-name}\rangle \longrightarrow \langle\textit{l-expression}\rangle$$

ToyL has a conventional assignment statement.

$$\langle\textit{statement}\rangle \longrightarrow \langle\textit{l-expression}\rangle\ \boxed{\texttt{:=}}\ \langle\textit{r-expression}\rangle\ \boxed{\texttt{;}}$$

⟨*l-expression*⟩ stands for "left-hand expression," meaning just the sort of expression that can be on the left-hand side of an assignment statement. Besides simple identifiers, this includes array references and record field references. An l-expression may also involve an identifier in the public part of a package, written *packagename.identifier*.

$$\langle\textit{l-expression}\rangle \longrightarrow \textit{identifier}$$

$$\langle\textit{l-expression}\rangle \longrightarrow \langle\textit{l-expression}\rangle\ \boxed{\texttt{[}}\ \langle\textit{r-expression}\rangle\ \boxed{\texttt{]}}$$

$$\langle\textit{l-expression}\rangle \longrightarrow \langle\textit{l-expression}\rangle\ \boxed{\texttt{.}}\ \langle\textit{l-expression}\rangle$$

An identifier is associated with a type according to its declaration. Array subscripting produces the component type of the array. Field references in records produce the field type.

$\langle$*r-expression*$\rangle$ stands for "right-hand expression," meaning just the sort of expression that can be on the right-hand side of an assignment statement.

$\langle$*r-expression*$\rangle$ $\longrightarrow$ $\langle$*l-expression*$\rangle$

$\langle$*r-expression*$\rangle$ $\longrightarrow$ $\langle$*constant*$\rangle$

$\langle$*r-expression*$\rangle$ $\longrightarrow$ $\langle$*function-call*$\rangle$

$\langle$*function-call*$\rangle$ $\longrightarrow$ $\langle$*func-name*$\rangle$ $\boxed{\texttt{(}}$ [{$\langle$*r-expression*$\rangle$ $\boxed{\texttt{,}}$} $\langle$*r-expression*$\rangle$] $\boxed{\texttt{)}}$

Constants include numeric constants, as well as the boolean constants `true` and `false`.

$\langle$*constant*$\rangle$ $\longrightarrow$ *number*

$\langle$*constant*$\rangle$ $\longrightarrow$ $\boxed{\texttt{true}}$ | $\boxed{\texttt{false}}$

ToyL permits conditionals in right-hand expressions, using a syntax almost like that of conditional statements. Note that the *else* clause of a conditional expression is *not* optional.

$\langle$*r-expression*$\rangle$ $\longrightarrow$ $\boxed{\texttt{if}}$ $\langle$*r-expression*$\rangle$ $\boxed{\texttt{then}}$ $\langle$*r-rexpression*$\rangle$
{$\boxed{\texttt{elsif}}$ $\langle$*r-expression*$\rangle$ $\boxed{\texttt{then}}$$\langle$*r-expression*$\rangle$}
$\boxed{\texttt{else}}$ $\langle$*r-expression*$\rangle$
$\boxed{\texttt{end}}$ $\boxed{\texttt{if}}$

Right-hand expressions can also be constructed from unary and binary operations, described below.

$\langle$*r-expression*$\rangle$ $\longrightarrow$ $\langle$*r-expression*$\rangle$ *binop* $\langle$*r-expression*$\rangle$

$\langle$*r-expression*$\rangle$ $\longrightarrow$ *unop* $\langle$*r-expression*$\rangle$

This highly ambiguous expression grammar can be resolved using precedence and associativity rules. The arithmetic operations are shown in this table.

Operation	argument type	result type	strength
`*`, `/`, `%`, `-` (unary)	integer	integer	6
`+`, `-`	integer	integer	5

All binary arithmetic operations are left-associative. Note that unary minus binds as tightly as multiplication. All of the above operations may appear in static (compile-time) expressions, which means in particular that they may appear in array bounds expressions.

The relational operations bind more weakly than arithmetic operations, and are non-associative. The logical operations bind most weakly of all operations, and the binary logical operations are left-associative.

Operation	argument type(s)	result type	strength
`<`, `>`, `>=`, `<=`,	integer	boolean	4
`=`, `<>`	integer, boolean	boolean	3
`not` (unary)	boolean	boolean	2
`and`	boolean	boolean	1
`or`	boolean	boolean	0

Of course, parentheses may also be used to group expressions.

$$\langle \textit{r-expression} \rangle \longrightarrow \boxed{\texttt{(}}\ \langle \textit{r-expression} \rangle\ \boxed{\texttt{)}}$$

Finally, ToyL provides two built-in procedures for input/output.

procedure get (x: out integer);
procedure put (x: in integer);

This ends the discussion of the syntax and semantics of ToyL.

Your task in this project is to write a recursive-descent parser to recognize the syntax of ToyL programs. You may adapt the given context-free grammar as necessary to resolve any ambiguities, precedences, associativities, etc, or to simplify the parser. Given a syntactically-correct ToyL input program, your ToyL parser should write the grammar rules of a leftmost derivation to an output file. However, if the ToyL input program contains any lexical or syntax errors, then your ToyL parser should write one or more appropriate error messages to standard output, for example: "Reserved word 'end' is not expected here." It is not necessary to report other kinds of errors (such as type errors).