CSU33071: ToY Compiler Project

Due: 15 Apr 2022

1 The ToY Language

You need to implement the parser and type-checker for the language ToY, defined below. Your implementation should be using the tools *flex* and *bison*. You are free to choose a programming language you are comfortable with that support *flex* and *bison*, so examples of supported languages are have C, C++ or Java.

For each of the parts of this project you need to submit the necessary files and basic instructions to compile and produce an executable from your code. Please do not submit files that cannot be compiled (pdfs, images of your code, etc.).

The language ToY is a simple imperative language with procedures and has many similarities with many similarities with C.

Syntax

The following tokens are defined:

- reserved words: `bool` `int` `true` `false` `void` `printf` `string` `struct` `if` `then` `else` `for` `return`
- Identifier `<id>`: a sequence of one or more letters/digits/underscores, starting with a letter or underscore and cannot be a reserved word. Identifiers are case-sensitive in ToY
- Integer literals: a sequence of one or more digits, potentially starting with the minus symbol. ToY does not have float numbers. Integers are 16 bit ranging from -32768 to 32767.
- String literals: a sequence of zero or more characters, written between double quotation marks (`"this is a string"`). ToY does not support escaped characters so `"Hello \"world\"!"` is invalid.
- Symbols: The symbols used by ToY are the following

  { } ; < > == <= >=
  != ! ( ) + - . =

- Comments: Text starting with `//` or `##` up to the end of the line. We don't support multi-line comments in our language.
- Whitespace: spaces, tabs, newline are all accepted. These are ignored by the language (except inside string literals).

NOTE: In the following `<x>,...` means zero or more repetitions of `<x>`, separated by comma. Therefore, `<x>, <x>,...` means *one or more* repetitions of `<x>`, separated by comma.

1.1 Types

ToY has the basic data types `int`, `bool`, `string`.

```
<type> ::= int | bool | string
```

Procedures can return a value of the above type or no value at all (`void`).

```
<return-type> ::= <type> | void
```

ToY also allows the programmer to define record types (structs) with the syntax

```
<struct> ::= struct <id> { <declaration>, <declaration>,... }
```

Note that a struct has at least one declaration in it, and declarations are comma-separated. Declarations are given by the grammar:

```
<declaration> ::= <type> <id>
```

A struct should not contain two declarations with the same identifier name.

Procedures

A ToY procedure definition has a unique identifier, zero or more arguments, given as a comma-separated list of declarations, and a return type.

```
<proc> ::= <return-type> <id> ( <declaration>,... ) { <statement> }
```

Statements

ToY statements follow the grammar:

```
<stmt> ::= for (<id> = <expr>; <expr> ; <statement>) <statement>
        |  if (<expr>) then <statement>
        |  if (<expr>) then <statement> else <statement>
        |  printf (<string>);
        |  return <expr>;
        |  { <statement-seq> }     #  compound statement
        |  <type> <id>;            #  variable declaration
        |  <l-exp> = <expr>;       #  assignment
        |  <id>(<expr>,...);       #  void procedure call
        |  id = <id>(<expr>,...);  #  non-void procedure call

<statement-seq> ::=                #  empty sequence
                 |  <stmt> <statement-seq>

<l-exp> ::= <id> | <id> . <l-exp>
```

All statements, with the exception of for-, if-, and compound statements are terminated with a semicolon. Compound statements contain a (potentially empty) sequence of statements. Variable declaration introduces new local variables at any point inside a procedure.

The first statement appearing inside a `for` construct is optional, similar to modern programming languages.

The else branch inside an if-then-else statement is optional. In programs such as the following, where there is an ambiguity as to which of the two if-statements has an else branch, you should resolve the ambiguity by considering that the else branch belongs to the *last* if-statement. See lecture and book about shift-reduce and reduce-reduce conflicts.

```
if e1 then if e2 then {} else {}
```

ToY does not allow declaration and assignment to be on the same line. This means that in our program `bool x = 0;` is represented as

```
bool x;
x = 0;
```

The grammar for l-expressions (this name is given because it appears on the left-hand side of assignment) allows us to refer to variables, as well as specific fields within variables of record type. As records can contain fields of other record type (see example in the following section), we need to allow for referencing fields of arbitrary nesting level.

Variables should be used (in l-expressions) only after they have been declared. If variables are declared inside a compound statement then they cannot be used outside of it. Similarly if they are declared inside a for- or if-statement.

Program Structure

A ToY source file contains a sequence of structure and procedure declarations, in any order, with at least one procedure declaration.

```
<pgm> ::= <proc>
        | <proc> <pgm>
        | <struct> <pgm>
```

No two declaration should have the same identifier. Also no two procedures should have the same name. Procedures can refer to each other no mater where they are being defined in the file (mutually recursive) and they can refer to themselves (recursive). An executable ToY program should have exactly one procedure with the following signature

```
void main() { ... }
```

Record types, defined with struct declarations, can be referred to in the program at any point where a type is needed, but only after the point of their declaration. Therefore ToY does not allow self-referential or cyclic definitions of record types. It does allow however multi level record definitions such as the following:

```
struct Name {
  string first_name,
  string last_name
}
struct Employee {
  Name name
}
struct Dept {
  Employee head
}
```

Expressions

ToY expressions can be the standard integer and boolean expressions, data literals and variable identifiers. Note that expressions do not include procedure calls, which are statements.

```
<exp> ::= <int-literal>
        | <string-literal>
        | true
        | false
        |  <exp> <op> <exp>
        | - <exp>
        | ! <exp>
        | <l-exp>
        | ( <exp> )
<op>  ::= + | - | * | / | mod | and | or | not | == | > | < | >= | <= | !=
```

Expressions (and other parts of the ToY language) allow for programs that do not make much sense. For example we could write

```
  "abc" / 4
```

Some of these programs can be ruled out by making the grammar of your parser more type-aware (e.g. defining separate int-exp, string-exp, bool-exp) or by delegating this task to the type-checker. No matter which option you choose, ToY is a statically type language and will always need a type-checker. Can you think of type-checks that cannot be easily implemented by the grammar?

Precedence and Associativity

The following precedence and associativity rules should apply for your parser:

- The dot operator is left associative.

- The relational and equality operators (e.g. $<$ and $==$) are non-associative (i.e., expressions like $a < b < c$ are not allowed and should cause a syntax error).
- All of the other binary operators are left associative.
- The unary minus and negation (!) operators have the highest precedence, then addition and subtraction, then multiplication and division, and then the relational and equality operators.

2 Marking Scheme

1. (20 points) **[Due 20 Feb]** Write a Lex Analyser for the language *ToY* defined below. The lexer should **only** print `VALID` OR `ERROR` to state if the program is valid or not. *For example*, the lexer should print `ERROR` for the following violations:
   (a) Bad string literals e.g., `"Hello`
   (b) Integers larger than `Integer.MAX_VALUE`
   (c) String containing escaped characters e.g., `"Hello \"world\"!"`
   (d) Invalid integer/ID e.g., `1wd7`
2. (30 points) **[Due 20 Mar]**Write a Parser for the language *ToY*. The parser should **only** print `VALID` OR `ERROR` to state if the program is valid or not according to the grammar described above. The parser should print `ERROR` *for example* for the following violations:
   (a) Reserve words used as variable names
   (b) Use of undeclared variables
   (c) Re-declaration of the same variable
   (d) Incorrect precedence and associativity rules are implemented e.g., `a < b < c`.
   (e) Bad struct usage: LHS of dot in variable reference is not a known struct and RHS is not a field in that struct.
3. (30 points) **[Due 10 Apr]** Extend the compiler to do some type-checking for `expr`. Once again, the parser should **only** print `VALID` if the program type-checks, `ERROR` otherwise. The compiler should print `ERROR` ***for at least*** the following checks:
   (a) Return type matches the method definition
   (b) Second component of the for loop is a boolean expression
   (c) If condition is a boolean expression
   (d) Expressions are well-typed i.e., `true + 1` and `!4` are invalid but `(1+3) + 4` is.
   (e) Assignments and function calls are well-typed.
   (f) Variables, procedures and records are used correctly (they are in scope and used at the correct type) according to the rules described above.

3 Submission Guidelines

You are to submit your lexer (`toy.l`) and parser (`toy.y`) code along with a Makefile that takes as input a filename through the command line. The file will contain example program to test your implementation.

Your project will be marked by compiling and running your code against a set of input examples. Please ensure

1. Any submitted code compiles and takes a single command line argument which is the name of a file that contains an input program in the ToY language.

2. An appropriate makefile is provided with the submission. The code should compile by typing "make" in the command line, at the root directory of your submission.

3. A test suite of input ToY programs are provided which test the capabilities of your compiler. Although no specific marks are assigned to this test suite, submitting a well-structured, working, and extensive test suite will allow you to claim more partial marks in the case when your compiler has errors.

4. Sharing test input programs between teams is **encouraged**; sharing code is **forbidden**!

Running Flex & Bison

To run the lexer & Parser together from your terminal.

For java:

```
$ jflex toy.l          # Compile Flex code to generate Java File
$ bison toy.y -L java # Compile Bison code to generate Java File
$ javac *.java         # Compile the  generated Java classes
$ java -cp ToY $@      # execute the parser
```

For C++:

```
$ bison toy.y                    # Compile Bison code to generate Java File
$ lex toy.l                      # Compile Flex code to generate Java File
$ gcc toy.tab.c lex.yy.c -ll     # Compile the generated C++ class
$  ./a.out $@                    # execute the parser
```

4 How to work in Pairs

An excellent way to work is to do *pair programming*. This helps improve design, coding, debugging and testing. When this is not possible and you decide to divide the work, you are strongly encouraged to still try and meet frequently, exchange code as often as possible to catch inconsistency early. An example of how to divide work would be

- Together decide the overall structure of the program and tests.
- Divide tokens into two parts and each partner is responsible for coding and testing their respective half of the tokens.
- Remember, it is still the pair responsibility to ensure the code produced is correct so it might be worth the other partner checking the completeness of the tests.
- Together try and combine the two sets of token and test the overall program

Whichever way you decide to work, do start early and build/test your code incrementally and frequently.