Project 4: Toyger to x86-64

Due December 3rd, 2021, 11:59pm

extra credit available for early submissions!

This is to be an individual effort. No partners.
No late work allowed after 48 hours; each day late automatically uses up one of your tokens. Late penalty applied if you run out of tokens.

1. Overview

For this assignment, you will be writing a compiler that can translate Toyger programs into x86-64. We are going to make lots of simplifying assumptions for this project - be sure to read this document carefully! I strongly suggest you deal with the different issues in the order I discuss them in Section 3. For each, test completely, including running the generated x86-64 program to see if the generated assembly program is correct (both syntactically and in terms of the underlying computation).

NOTE:

- Your scanner must be constructed by Lex. We are using the same lexical definition of Toyger as Project 2/3. You can re-use your lex specification for this assignment. You may need to augment the scanner to provide support needed in translation.
- Your parser must be an LR parser constructed by YACC. We are using the same Toyger grammar from Project 3. You can re-use your parser from P3 for this assignment. You will need to augment the parser to implement the translation.
- Make sure that your code compiles and runs with specified tool on **zeus.vse.gmu.edu**; programs that do not compile and run on zeus will not receive much credit.

2. Input/Output

Input: Toyger

The source programs that your compiler will take as input are written in Toyger. The lexical definition, grammar, and semantic rules of Toyger are included in the Appendixes of this document for your reference. You can assume the following features on the input to simplify the task:

- **All input programs are free of syntax errors and semantic errors;**
- **All functions have six or fewer parameters and all parameters are integer type;**
  - **The only exception is the built-in function printstring, which has a string parameter.**
- **Functions either have an integer return or no return (void);**
- **No recursive functions;**
- **We only have integer variables;**
- **Type inferences will not be needed -- variable declarations will always specify a type;**
- **Assume integers are 4-bytes long (i.e. long word (l) size in x86-64).**

Output: x86-64

The target programs generated by your compiler must be legal x86-64 programs. You only need to use a subset of x86-64 instructions in the translation. Chcek the sample outputs in the handout package to get started. We will use **gcc** on **zeus.vse.gmu.edu** to test whether your output can be assembled and run with the expected behavior. You should do the same for your own testing.

3. Translation

I strongly suggest you work on this assignment incrementally, testing each added part thoroughly before moving to the next. I'm going to walk through the parts in the order that I implemented them - this order is not required but might make your life easier.

There are usually multiple ways to translate the source program into a valid and working target program. The x86-64 translation included in this document and handout are just examples. You do **not** need to generate exactly the same code and you do **not** need to optimize your generated code either. Pay close attention to some specific requirements we have for this assignment in the following discussion.

3.1 Setup and Exit

For this assignment, it is required that the generated x86-64 program has a global label `main`. It will be used as the starting point of the execution. At the end of the main function, we need to return from main() to terminate the execution. It's also recommended to use the stack to save and restore the frame pointer (%rbp). The typical x86-64 code for the start and exit action is given as below. If there is a return, additional code to set the return register (%eax) will be needed before return.

x86-64 (start):

```
.text
.globl main
main:
  pushq %rbp
```

x86-64 (exit):

```
.text
  popq %rbp
  ret
```

3.2 Output Functions

Implementing print functions first will make it easier to test your code by running it. You will generate code for the built-in functions `printint()` and `printstring()`. You can call library function printf to support those. Implementation notes and hints:

- In order to call printf, set register %rdi/%edi with the address of the format string, set %rsi/%esi with the value to be printed, and set %rax/%eax to be 0.
- Start with basic printing using constant values only.
- You will need to generate code to load constant into a register when factor→NUMBER or factor→STRING_LITERAL. The register used needs to be passed up the parse tree so the print statements can copy the value into the %rsi/%esi to call printf.
- To print string constants, you have to create the string before you can print it. Be sure that the label used for the string is unique.
- Example translations:

Toyger:

```
printint(440)
```

x86-64:

```
.section .rodata
LPRINT0:
  .string "%d\n" #format string
.text
movl $440, %r10d
movl $LPRINT0, %edi
movl %r10d, %esi
xorl %eax, %eax
call printf #print int 440
```

Toyger:

```
printstring("This is a test")
```

x86-64:

```
.section .rodata
LPRINT1:
  .string "%s\n" #format string
str8:  .string "This is a test"
.text
movl $str8, %r10d
movl $LPRINT1, %edi
movl %r10d, %esi
xorl %eax, %eax
call printf #print str8
```

3.3 Arithmetic Expressions

Next, work on generating x86-64 code for basic arithmetic expressions that use only constant values. **Lecture 7 includes useful references for what need to be done**.

- At the leaves of a parse tree, you need to put values into registers (similar to print).
- You need to have an attribute for some non-terminal symbols (e.g. expr) that keeps track of what register is holding the intermediate value.
- At the internal nodes, you need to generate code that would perform the computation using the registers holding intermediate values.
- Do not generate additional labels to hold intermediate values. Use registers for those. Check below for general guidelines regarding register allocation.
- If the expression is used in a print statement, the sequence will end with a sequence leading to the call to printf like the example below. Other statements may need to have different usages of the expression value, which you should be able to find in the assigned register.

The example below illustrates the translation of an expression with multiple subcomponents.

```
Toyger:
 printint(5*8 - 3*2)
```

```
x86-64:
movl $5,%r10d
movl $8,%r11d
imull %r11d, %r10d
movl $3, %r11d
movl $2, %r12d
imull %r12d, %r11d
subl %r11d, %r10d   #expr value in %r10d
movl $LPRINT0, %edi
movl %r10d, %esi
xorl %eax, %eax
call printf
```

Don't worry about generating the most efficient code. That is much more difficult and doesn't fix anything a good optimizer wouldn't fix. Also, don't worry about generating code identical to the examples in this document or the handout package. As long as the computation gets the same result, your answer is fine.

Register usage guidelines. Use temporary registers to hold intermediate results. Because the number of registers is limited, you need to keep track of which ones you are actually using at any given point in execution. You have 15 registers available but it is safest to only use a subset, saving the other registers like %eax and the parameter registers for special use. The subset of 32 bit registers we recommend you to use is {%r10d, %r11d, %r12d, %r13d, %r14d, %r15d}. Assuming you "allocate'' a register when you need one and ``free'' a register when you are done with it, this is plenty for any example that we will use for testing/grading. You will need a data structure that tells you what registers are available (initially, all registers are available). Write two functions: one that gets an available register and another that returns a register that you are no longer using. Returning to the simple example above, if we start out with all registers available, we can keep track of the following:

Statement	Registers Available after the statement	Registers In Use
Initially	{%r10d-r15d}	{}
movl $5,%r10d	{%r11d,%r12d,%r13d,%r14d,%r15d}	{%r10d}
movl $8,%r11d	{%r12d,%r13d,%r14d,%r15d}	{%r10d, %r11d}
imull %r11d,%r10d	{%r11d,%r12d,%r13d,%r14d,%r15d}	{%r10d}
movl $3, %r11d	{%r12d,%r13d,%r14d,%r15d}	{%r10d, %r11d}
movl $2, %r12d	{%r13d,%r14d,%r15d}	{%r10d, %r11d, %r12d}
imull %r12d,%r11d	{%r12d,%r13d,%r14d,%r15d}	{%r10d, %r11d}
subl %r11d,%r10d	{%r11d,%r12d,%r13d,%r14d,%r15d}	{%r10d}

```
movl $LPRINT0, %edi      {%r11d,%r12d,%r13d,%r14d,%r15d}          {%r10d}
movl %r10d, %esi         {%r10d,%r11d,%r12d,%r13d,%r14d,%r15d}    {}
```

Hint: You need to look for rules where information in multiple registers is used to compute something and where the result is left in one register. For example, in production `exp : exp + term`, you need to be sure to free one register because your generated code will add the contents of two registers and put the result in a single register. One of the registers is no longer needed.

3.4 Global Variables and Assignments

Global variables can be supported using static allocation. When parsing declarations, each ID gets allocated a unique label in a .data section. Note the in x86-64 assembly programs, multiple .data section are allowed and they can be placed in an interleaved fashion with multiple .text sections. Late references to the ID can use that unique label with %rip to perform instruction-pointer-relative addressing. Check examples below.

Toyger:
```
var x:int !global x
```

x86-64:
```
.data
global_x: .long 0
```

Toyger:
```
x := 5 !global x
```

x86-64:
```
.text
movl $5, %r10d
movl %r10d, global_x(%rip)
```

If the RHS of the assignment statement is a more complicated expression, you should have code generated to evaluate its value and a register allocated to hold the value when process the expression. The translation of the assignment statement needs to combine the address from LHS ID and the register from RHS expression in a mov instruction. **Note:** we only have integer variables in this project. It is possible to accommodate global variables in the stack (AR of main). But all our examples are using static allocation (labels) for global variables.

3.5 Input

The `getint()` function only allows reading in integers. The translation is similar to assignments but will need to use the library function scanf. Implementation notes and hints:

- In order to call scanf, set register %edi with the address of the format string, set %esi with the address to be updated by scanf, and set %eax to be 0.
- Example:
  - Only the red lines are generated when we parse the input statement. The data declarations should be generated when we set up and parse the declaration of x.

Toyger:
```
x := getint()!global x
```

x86-64:
```
.section .rodata
.LGETINT:
  .string "%d"
.data
global_x: .long 0 #label for x
…
.text
   movl $LGETINT, %edi
   movl $global_x, %esi #load address to be updated by scanf
   xorl %eax, %eax
   call scanf #read in an int to update global_x
```

3.6 Control Flow

You need to generate the appropriate labels, branches, and jumps for the control flow expressions of Toyger. **You can find references in Lecture 7**. One way to deal with the fact that control flow statements can be nested is to use the YACC stack to hold onto the labels associated with each loop or if statement. Labels can be associated with terminal symbols of the production. An alternative implementation is to use a stack onto which you push active labels. Once you finish with the particular statement, pop off the associated labels. Remember that control flow statements use registers and they need to be freed once you are done with them. Examples:

Toyger:

```
   if (5>3) then
      x := 10
   end
```

x86-64:

```
      movl $5, %r10d
      movl $3, %r11d
      cmpl %r11d,%r10d
      setg %al
      movzbl %al, %eax
      cmpl $0, %eax
      je L0
      movl $10, %r10d
      movl %r10d, global_x(%rip)
L0:
```

Toyger:

```
   for x:=1 to 10 do
      y := y + 1
   end
```

x86-64:

```
      movl $1, %r10d
      movl $10, %r11d
      movl %r10d, global_x(%rip)
L0:
      cmpl %r11d, %r10d
      jg L1
      movl global_y(%rip), %r12d
      movl $1, %r13d
      addl %r13d, %r12d
      movl %r12d, global_y(%rip)
      incl %r10d
      movl %r10d, global_x(%rip)
      jmp L0
L1:
```

3.7 Procedure Call/Return Control Flow

Implementing the control flow for procedure call/return is relatively straightforward assuming the procedure has no parameters. You must use `call` and `ret` to manage control flow. This is also a good time to add code to save and restore registers and build an activation record for the calls. For this assignment, you can use either static allocation or stack allocation or a combination (your choice) for the activation record.

- For every procedure, your compiler needs to generate calling sequence (prologue and epilogue). This is the code inserted at the start and end of a function body to take care of saving machine status (shared registers) and initializing data if necessary.
- Calling sequence also needs to be inserted at every call site (instruction that makes a function call). Before a function call, the caller needs to save important machine status (registers for example. After a function call, the caller needs to restore the saved values.
- Remember to save registers before making a call. This applies to the calls to built-in functions scanf and printf as well. You can assume the entrance procedure main() does not have any parameter and skip the saving/restoration of registers around printf/scanf in main(). But those are required for all other functions that need to call printf/scanf.
- You can test control flow of procedure call/return using parameter-less functions with no local variables.

Example assuming **static allocation** of activation records:

```
.data
f_callee_regs:.zero 40    # storage for %rbx,$r12,%r13,%r14,%r15 (use stack for %rbp)
f_caller_regs:.zero 64    # storage for %rdi, $rsi,%rdx,%rcx,%r8,%r9,%r10,%r11
```

- Create labels for saved registers.

x86-64 prologue (at the start of a function):

```
pushq %rbp
movq $f_callee_regs, %rax
movq %rbx, (%rax)
movq %r12, 8(%rax)
movq %r13, 16(%rax)
movq %r14, 24(%rax)
movq %r15, 32(%rax)
```

x86-64 epilogue (at the return point):

```
movq $f_callee_regs, %rdx
movq (%rdx), %rbx
movq 8(%rdx), %r12
movq 16(%rdx), %r13
movq 24(%rdx), %r14
movq 32(%rdx), %r15
popq %rbp
ret
```

- Note: we are using quad word operations (movq) because we want to save all 64 bits of the relevant registers.
- We still use push/pop for %rbp to avoid violating the special rules of frame maintenance.

x86-64 pre-call:

```
movq $f_caller_regs, %rbx
movq %rdi, (%rbx)
movq %rsi, 8(%rbx)
movq %rdx, 16(%rbx)
movq %rcx, 24(%rbx)
movq %r8, 32(%rbx)
movq %r9, 40(%rbx)
movq %r10, 48(%rbx)
movq %r11, 56(%rbx)
// load parameters
call function
```

x86-64 after return:

```
call function
movq (%rbx), %rdi
movq 8(%rbx), %rsi
movq 16(%rbx), %rdx
movq 24(%rbx), %rcx
movq 32(%rbx), %r8
movq 40(%rbx), %r9
movq 48(%rbx), %r10
movq 56(%rbx), %r11
//use eax (return value)
```

Example assuming **stack allocation** of activation records:

x86-64 prologue (at the start of a function):

```
pushq %rbp
pushq %rbx
pushq %r12
pushq %r13
pushq %r14
pushq %r15
subq $128, %rsp
```

x86-64 epilogue (at the return point):

```
addq $128, %rsp
popq %r15
popq %r14
popq %r13
popq %r12
popq %rbp
popq %rbx
ret
```

- In the example above, we grow the stack for another 128 bytes after saving callee-saved registers. This additional space would be used for the 8 caller-save registers and local/temp data. You can assume a size of 128 bytes is sufficient for all functions that we will use in grading. You can make this number larger if you want, but do not go smaller.

x86-64 pre-call:

```
movq %rdi, 64(%rsp)
movq %rsi, 72(%rsp)
movq %rdx, 80(%rsp)
movq %rcx, 88(%rsp)
movq %r8, 96(%rsp)
movq %r9, 104(%rsp)
movq %r10, 112(%rsp)
movq %r11, 120(%rsp)
// load parameters
call function
```

x86-64 after return:

```
call function
movq 64(%rsp), %rdi
movq 72(%rsp), %rsi
movq 80(%rsp), %rdx
movq 88(%rsp), %rcx
movq 96(%rsp), %r8
movq 104(%rsp), %r9
movq 112(%rsp), %r10
movq 120(%rsp), %r11
//use eax (return value)
```

3.7 Procedure Call/Return Data Flow

Parameters and return values. Parameters are passed into our functions using registers (%rdi/edi, %rsi/esi, %rdx,edx, %rcx/%ecx, %r8/%r8d, %r9/%r9d). Additional parameters would go onto the stack - however you will only be tested on 6 or fewer parameters. The result of a function is always put into %rax/%eax. You can see this pattern in the scanf and printf calls above. You can assume integer parameters and integer return only in our test cases. Integer parameters are passed by value. Since we aren't using the parameter registers to do any of the work described above (except for the epilogue example above), you can use these registers directly in your generated function code, rather than copying the parameters in activation record and using them from there (which also works).

Local variables. All local variables in a function should be saved in the activation record. Again, you can choose to implement using static allocation or stack allocation.

- Static allocation
  - Assign each local variable a unique label **L** at the declaration point
  - Accesses to the variable can be translated using **L(%rip)** where **L** is the label
    - For example, if there are two local integer variables x and y, and we have created two labels f_x and f_y. Then we can access x with f_x(%rip) and y with f_y(%rip).
- Stack allocation
  - Decide an offset **D** for each variable at the declaration point
  - Accesses to the variable can be translated using **D(%rsp)** where **D** is the offset
    - For example, if there are two local integer variables x and y, x might live at 0(%rsp) and y might live at 4(%rsp).
  - Our earlier example allocates 128 bytes for each stack frame. These bytes are used to save (caller-save) registers and local variables. You can assume that there will be no more than 10 local integer variables in a function, so the frame is probably too large but better safe than sorry.

Symbol table. You will need to use a symbol table just as with Project 3; however, the important information is not the type, but information that will tell you whether every identifier is global, local or a parameter and where to find them, so that you can generate the correct code to for the given identifier.

4. Input/Output

Similar as before, your compiler should be reading from standard input (`stdin`) and output the target program to standard output (`stdout`). Since we assume that all input programs are well-structured Toyger programs with no lexical/syntax/semantic errors, no other output/error messages are expected.

Once you have the x86-64 code generated, make sure you assemble it with gcc on zeus. It should be assembled without any assembler errors or warnings. Then you can run the generated executable to check whether the translation is correct. We prepare a group of test cases in the handout package, including the Toyger input and the corresponding x86-64 output. You can find the sample files under tests/ folder. A readme.txt file in that folder provides a brief description of each input. Make sure you write your own test cases and thoroughly test your compiler.

5. How to Compile/Run

Similar as before, a starting package is provided to you as your *optional* starting point (**`p4_handout_c.tar`** or **`p4_handout_java.tar`**). Once untarred on Zeus, it will create a folder for you which is similar to the starting package of Project 3. You can swap in your lex/yacc specifications from P3 ***with names updated***:

- You can type "make" to compile and generate the executable.
  - For C package, the generated executable is **p4_c**
  - For Java package, the generated driver class is **p4_java.class**
- Both C and Java starter code has the Toyger basic scanner/parser implemented.
- All example files are under the tests folder. Check tests/readme.txt to get more details.

Make sure your parser follows the input/output requirements as described above. Then you can use the commands as the examples below to run the parser and generate the x86-64 file (.s):

```
zeus-1:~/cs440/projects/p4>./p4_c <test0.tog >test0.s
zeus-1:~/cs440/projects/p4>java p4_java <test0.tog >test0.s
```

We will be checking your generated output for grading. More test files than the provided ones will be used for grading. Make sure you write your own test cases to cover different scenarios.

Testing with the generated assembly files. One important step in testing is to assemble the generated x86-64 file with gcc and run the executable. To make the testing easier, we include a simple script test.sh in the starting folder (start_c/ and start_java/). To test your compiler with the input testN.tog, use the following command:

```
zeus-1:~/cs440/projects/p4>./test.sh N
```

6. What / How to Submit

When you are ready to turn in, follow the instructions below to make the submission to Gradescope. Requirements for your submission:

- Submit to Gradescope. A link to Gradescope is available from Blackboard.
- Make sure your code is well-documented as always.
- You must include all source files but no executable or intermediate files.
- You must include a **Makefile** that can generate the executable on zeus.
- In the submission, make sure we can use the Makefile to generate an executable with the **name** as follows:
  - If your working language is C/C++, the generated executable must be **p4_c**
  - If your working language is Java, the generated driver class file must be **p4_java.class**
- You must put all files in a folder named **p4_username** and then create a **p4_username.tar** for the folder.
  - Here username is the first part of your GMU email address.
  - For example: p4_yzhong/ and p4_yzhong.tar
  - The submitted file should take this structure:

  p4_yzhong.tar --> p4_yzhong --> Makefile
  --> p4_c.l
  --> p4_c.y
  -->…
- Your program must compile and work on **zeus.vs.gmu.edu** and work with the JFlex posted on Blackboard (if you use Java). No need to include the jar file in your submission.
- **Cheating is not permitted in any form. We will enforce the university honor code.**

7. Grading Rubric

- Code submission and documentation 20%
  - Make sure you follow submission instructions including file names
  - Make sure your code is well commented
- Code can be compiled w/o any error 10%
  - Make sure to include a Makefile to compile your code on zeus
- Toyger translation 75%
  - Start/Exit/Output constants (10%)
  - Arithmetic expressions with constants (10%)
  - Input and output with variables (10%)
  - Assignments (10%)
  - Branch statements (10%)
  - Loops (10%)
  - Function calls (15% including 5% extra credit)
- **Notes:**
  - This assignment has 105 total available points with 5 points as extra credit.
  - Points will be taken off if your translation is not consistent with our requirements on the generated x86-64 code even if they are valid x86-64 programs.
- **Extra credit for early submissions:**
  - 1% extra credit rewarded for every 24 hours your submission made before the due time
  - Up to 3% extra credit will be rewarded
  - Your latest submission will be used for grading and extra credit checking. You CANNOT choose which one counts.

Appendix I: Toyger Lexical Specification

Toyger tokens include **keywords, punctuation elements, operators, IDs, integers, strings,** and **comments**. They are defined as the following:

- **Keywords:** `let in end var function printint printstring getint return if then else for to do int string void`
- **Punctuation elements:** `( ) : , = ;`
- **Operators:**
  - Arithmetic: :=(assignment) `+ - * /`
  - Comparison: `==` (equality) `< <= > >= <>` (not equal)
- **Identifiers:** any non-empty sequence of letters (a-z and A-Z), digits (0-9), and _ (underscore) starting with a letter. We will use **ID** as the token name.
- **Numbers** (integers only)**:** any non-empty sequence of digits (0-9) with no redundant leading zeros. We will use **NUMBER** as the token name.
- **Strings:** a possibly empty sequence of characters between (and including) the closest pair of quotation marks ("). A string cannot spam more than one line. A literal quotation mark can be included as part of a string using the C-style escape format (\"). We will use **STRING_LITERAL** as the token name.
- **Comments:** start with double slash `//` until the end of the line (`//` in strings loses its special meaning).

Note:

- Toyger is <u>case-sensitive</u>.
- White-spaces (space, tab, newline) are allowed in input and work as delimiters between tokens.
- For comments, your scanner should recognize them but not report them to the parser -- similar to what the scanner needs to do for white-spaces.

Appendix II: Toyger Syntax Specification

A Toyger program consists of declarations followed by a sequence of statements. **Notes**:

- In the grammar below, operators and punctuation elements are used directly (e.g. : and =).
- Keywords are underlined.
- Other token names are capitalized (e.g. ID).
- `program` is the start symbol of the grammar.
- Spaces shown in grammar rules are not required from the input – they are inserted to make the productions easier to read.
- Comment is not included in the grammar since it can be placed between any two legal tokens. Your scanner should recognize them but there is no need to report them to the parser.
- The language that defined here for this assignment is actually a subset of what we will work on in later assignments.

```
program         → let decs in statements end
decs            → dec decs | ϵ
dec             → var_dec | function_dec
var_dec         → var ID := expr | var ID : type
type            → int | string | void
function_dec    → function ID (params):type = local_dec statements end |
                  function ID ( ):type = local_dec statements end
local_dec       → let var_decs in | ϵ
var_decs        → var_decs var_dec | ϵ
params          → params , parameter | parameter
parameter       → ID : type
statements      → statements ; statement | statement
statement       → assignment_stmt | print_stmt | input_stmt |
                  if_stmt | for_stmt | call_stmt | return_stmt
assignment_stmt → ID := expr
input_stmt      → ID := getint()
return_stmt     → return expr  | return
call_stmt       → ID ( ) |  ID ( expr_list )
print_stmt      → printint ( expr ) | printstring ( expr )
rel_expr        → expr == expr |  expr <> expr |  expr < expr | expr <= expr|
                  expr > expr |  expr >= expr | (rel_expr)
if_stmt         → if rel_expr then statements end |
                  if rel_expr then statements else statements end
for_stmt        → for ID := expr to expr do statements end
expr            → expr + term | expr – term | term
term            → term * factor | term / factor | factor
factor          → ( expr ) | NUMBER | STRING_LITERAL | ID | call_stmt
expr_list       → expr_list , expr | expr
```

Appendix III: Toyger Semantic Rules

Scoping

A Toyger program consists of function/variable declarations followed by a sequence of statements. Every source file is associated with a global scope. Global names include all function names and variable names declared outside of functions. Each function introduces a separate local scope that is nested inside the global scope. Identifiers belongs to a local scope includes parameters and local variables of that function. No nested function definitions are allowed. Toyger uses static scoping.

- Global names (function/variable) are visible after the declaration point until the end of the file.
    - The sequence of statements after declarations can access all global names.
    - Global names that have been declared are visible inside functions unless they are hidden by a local variable with the same name. This implies that recursive functions are allowed but we do not support mutual recursion.
- Local names (variable only) are visible only within the function in which they are declared or introduced as parameters.

ID Definitions

Toyger IDs need explicit declarations. No names can be used before declared.

- A function cannot be called before it has been declared.
- A variable cannot be refenced if it has not been declared.
- Each name/ID can only be declared once *in each scope*:
    - It is not allowed to define multiple functions with the same name but different return type or different parameters.
    - It is not allowed to define a function and a global variable with the same name.
    - It is not allowed to declare a variable multiple times in one scope.
        - For a function, we cannot have a local variable and a parameter sharing the same name.
    - It is allowed to define a local variable with the same name as a global variable or function. The global name will be shadowed by the local name.

Types

Variables. Variables in Toyger can take two possible types: integer and string. We follow these rules:

- A variable can be declared to take a type; or
- A variable can be initialized with an expression. In this case, the type of the variable is determined by the type of that initial expression.
- You can assume that a variable will not be declared as void or initialized with an expression of type void. No need to check this in your code.
- Token NUMBER is of type integer and token STRING_LITERAL is of type string.

Functions. Function declaration defines the name, return type, and the list of parameters of a function. There are three possible return types: integer, string, and void (no return).

Special Note for Project 4:

- All variables/parameters are of integer type;
- Functions take no more than six parameters;
- Function return (if any) must be integer type;
- Variable declarations will always specify a type instead of using expression to initialize.

Instructions

The semantics of most instructions should be straightforward. The clarification for the loop (for_stmt) is as below:

for_stmt → for ID := $expr_1$ to $expr_2$ do statements end

Semantic as pseudocode using a while loop:

```
start_val = expr1;  //evaluating expr1
end_val = expr2;    //evaluating expr2
ID := start_val;    //initializing loop variable ID
while (ID <= end_val)
     statements; //loop body
     ID := ID+1;   //updating loop variable ID
```