Project Report: TL 16.0 Compiler

Rohit Mehra
Department of Computer Science
University of Texas at San Antonio
Email: gjw811@my.utsa.edu

Jinay Jani
Department of Computer Science
University of Texas at San Antonio
Email: glp214@my.utsa.edu

I. Purpose

The project is to design and develop a compiler for the "Toy Language", described in TL 16.0 specifications, which is derived from a simplified subset of Pascal. The compiler translates TL source code, producing output at several intermediate stages, into MIPS assembly code executable in a simulator. The project is developed as part of the Programming Languages and Compilers Course during Spring 2017. We implemented this compiler using Python 3.5 language and Pycharm IDE.

II. Architecture

A. Scanner

- The scanner expects a TL input file, <basename>.tl, to be specified as command-line argument when it is invoked. If the input file is not a syntactically-valid program (i.e., if the program contains a lexical element that is not a legal token or does not match the BNF grammar), the compiler outputs an error message to standard error stream.
- Our Scanner class is present in file **scanner.py** which implements tokenization using Regular Expressions. It also does a pre check to validate that the input program follows the rules of the TL 16.0 BNF grammar, using an external module named *pyparsing*. This step is optional.
- An instance of the Scanner class would store the program it got from the input file in its attribute named *program*. Tokens are represented using *Token* class which is also present in **scanner.py** file. When *parse* function of scanner the object is invoked the program is tokenized and the *token_objects* attribute of the scanner object is populated with the token objects and *tokens* attribute is populated with the string representation of the token objects. The string representation of the stream of token objects is output in the **<basename>.tok** file.

B. Parser and Type Checking

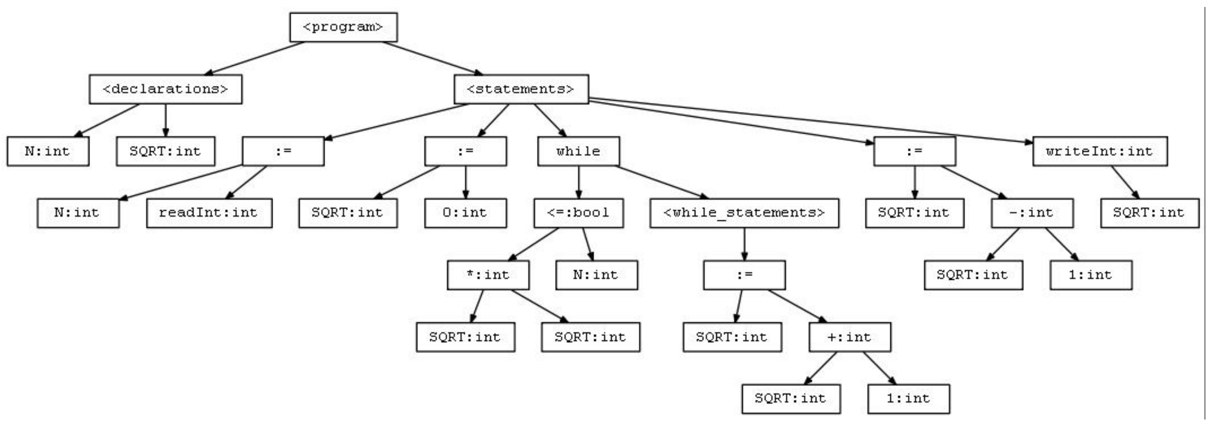


Figure 1: Square Root Program AST

- There are two parsers involved in this phase, Parse Tree Parser and Abstract Syntax Tree Parser. Both are independent of each other in our case. Parse Tree Parser present in **parsetreeparsing.py**, is just used to generate the parse tree and visualize it using **parsetree_visual.py**. On the other hand Abstract Syntax Parser present in **astparsing.py** is used to generate AST which is further used in next phase.
- AST Parser incorporates type checking with the help of a variable map dictionary (*variable_map* attribute of AST Parser). We implemented a visualization script **ast_visual.py** which is used to generate **<basename>.ast.dot** file, which is further used to render a **<basename>.ast.jpg** file containing type checked AST representation, this is done if graphviz application is installed in the system. If graphviz is not installed the **ast_visual.py** would just generate the DOT file.

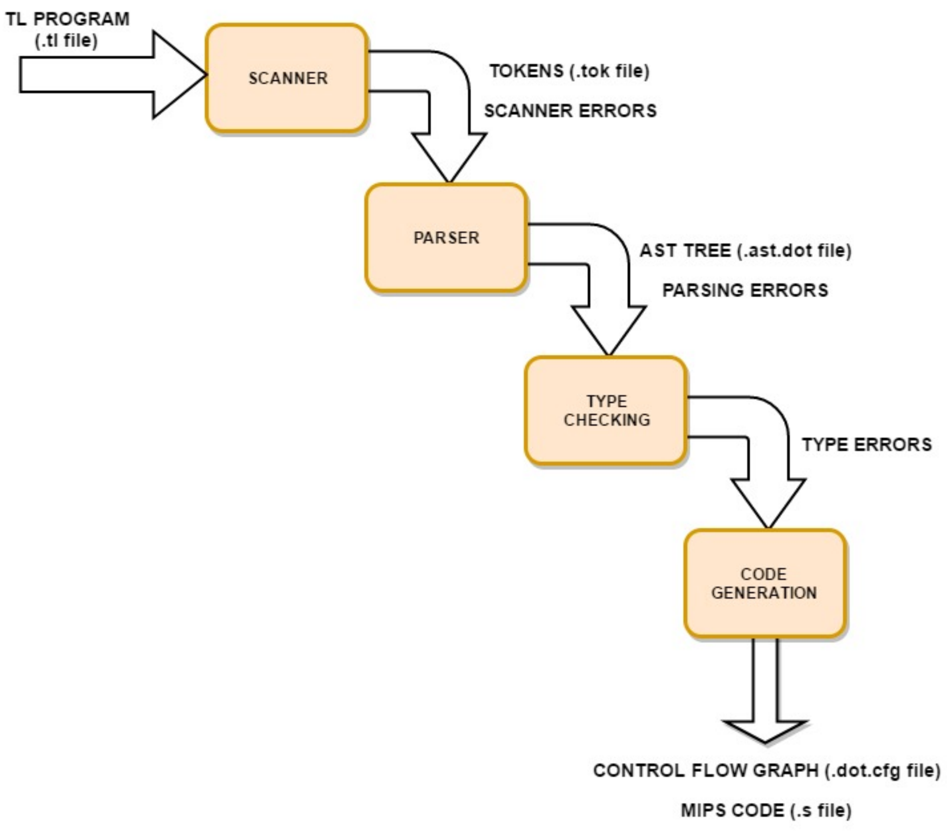
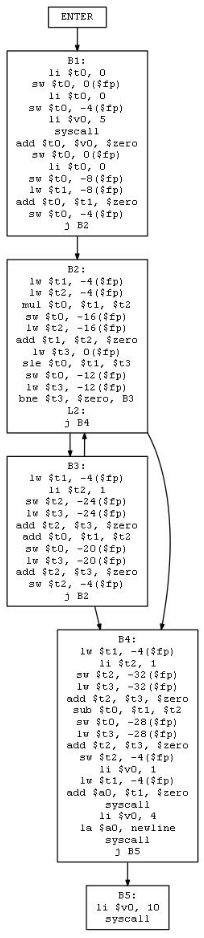


Figure 2: Compiler Architecture

C. MIPS Assembly and Control Flow Graph

- AST received by assembly generator script **asmgeneration.py** converts the Abstract Syntax Tree into assembly code directly, skipping the intermediate code generation step. The string representation of the MIPS code is output in the **<basename>.s** file.
- In parallel CFG is also generated using **main.py** in the form of DOT file with name representation as **<basename>.cfg.dot**, which is further used to render a **<basename>.cfg.jpg** file containing CFG representation, this is done if graphviz application is installed in the system. If graphviz is not installed the **main.py** would just generate the DOT file.


```
ENTER

B1:
li $t0, 0
sw $t0, 0($fp)
li $t0, 0
sw $t0, -4($fp)
li $v0, 5
syscall
add $t0, $v0, $zero
sw $t0, 0($fp)
li $t0, 0
sw $t0, -8($fp)
lw $t1, -8($fp)
add $t0, $t1, $zero
sw $t0, -4($fp)
j B2

B2:
lw $t1, -4($fp)
lw $t2, -4($fp)
mul $t0, $t1, $t2
sw $t0, -16($fp)
lw $t2, -16($fp)
add $t1, $t2, $zero
lw $t3, 0($fp)
sle $t0, $t1, $t3
sw $t0, -12($fp)
lw $t3, -12($fp)
bne $t3, $zero, B3
L2:
j B4

B3:
lw $t1, -4($fp)
li $t2, 1
sw $t2, -24($fp)
lw $t3, -24($fp)
add $t2, $t3, $zero
add $t0, $t1, $t2
sw $t0, -20($fp)
lw $t3, -20($fp)
add $t2, $t3, $zero
sw $t2, -4($fp)
j B2

B4:
lw $t1, -4($fp)
li $t2, 1
sw $t2, -32($fp)
lw $t3, -32($fp)
add $t2, $t3, $zero
sub $t0, $t1, $t2
sw $t0, -28($fp)
lw $t3, -28($fp)
add $t2, $t3, $zero
sw $t2, -4($fp)
li $v0, 1
lw $t1, -4($fp)
add $a0, $t1, $zero
syscall
li $v0, 4
la $a0, newline
syscall
j B5

B5:
li $v0, 10
syscall
```

Figure 3: Square Root Program CFG

III. Technologies Used

The whole compiler was coded in using Python 3.5 and Pycharm IDE. Other tools like Graphviz and QT simulator where used to check the AST, CFG and execution of the MIPS instructions.

IV. Input and Output Example

A. Fibonacci Program in TL 16.0

1. Program:

```
\% This program computes the Fibonacci Sequence
\% It Reads the highest value to count to from the user
\% It ouputs the sequence up to the user's limit

program
 var LAST as int ;
 var NEXTTOLAST as int ;
 var LIMIT as int ;
 var FIB as int ;
begin
  LIMIT := readInt ;
  LAST := 1 ;
  NEXTTOLAST := 0 ;
  FIB := 1 ;
  writeInt 0 ;
  while FIB <= LIMIT do
   writeInt FIB ;
   FIB := LAST + NEXTTOLAST ;
   NEXTTOLAST := LAST ;
   LAST := FIB ;
  end ;
 end
```

2. Tokens Generated:

```
PROGRAM
VAR
ident(LAST)
AS
INT
SC
VAR
ident(NEXTTOLAST)
AS
INT
SC
VAR
ident(LIMIT)
AS
INT
SC
VAR
ident(FIB)
AS
INT
SC
.
.
So on
```

3. AST Tree(Type Checked):

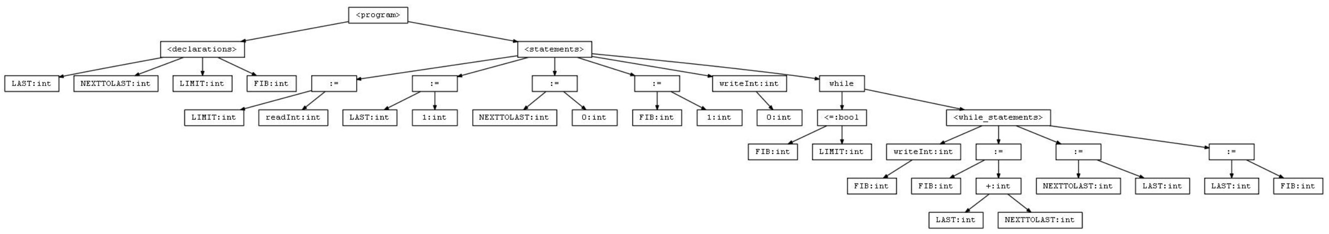
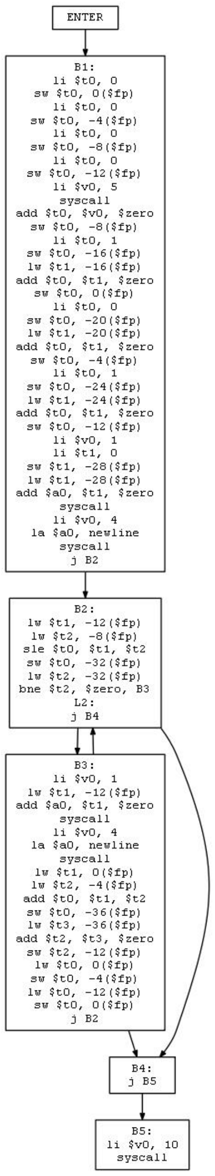


Figure 4: Fibonacci Program AST

4. Compiled Program and CFG:



```
ENTER

B1:
li $t0, 0
sw $t0, 0($fp)
li $t0, 0
sw $t0, -4($fp)
li $t0, 0
sw $t0, -8($fp)
li $t0, 0
sw $t0, -12($fp)
li $v0, 5
syscall
add $t0, $v0, $zero
sw $t0, -8($fp)
li $t0, 1
sw $t0, -16($fp)
lw $t1, -16($fp)
add $t0, $t1, $zero
sw $t0, 0($fp)
li $t0, 0
sw $t0, -20($fp)
lw $t1, -20($fp)
add $t0, $t1, $zero
sw $t0, -4($fp)
li $t0, 1
sw $t0, -24($fp)
lw $t1, -24($fp)
add $t0, $t1, $zero
sw $t0, -12($fp)
li $v0, 1
li $t1, 0
sw $t1, -28($fp)
lw $t1, -28($fp)
add $a0, $t1, $zero
syscall
li $v0, 4
la $a0, newline
syscall
j B2

B2:
lw $t1, -12($fp)
lw $t2, -8($fp)
sle $t0, $t1, $t2
sw $t0, -32($fp)
lw $t2, -32($fp)
bne $t2, $zero, B3
L2:
j B4

B3:
li $v0, 1
lw $t1, -12($fp)
add $a0, $t1, $zero
syscall
li $v0, 4
la $a0, newline
syscall
lw $t1, 0($fp)
lw $t2, -4($fp)
add $t0, $t1, $t2
sw $t0, -36($fp)
lw $t3, -36($fp)
add $t2, $t3, $zero
sw $t2, -12($fp)
lw $t0, 0($fp)
sw $t0, -4($fp)
lw $t0, -12($fp)
sw $t0, 0($fp)
j B2

B4:
j B5

B5:
li $v0, 10
syscall
```

Figure 5: Fibonacci Program CFG

V. Contributions

1. Rohit Mehra:

Parser, Code Generation Phase, Testing codes on fox server and Final Report.

A. Jinay Jani:

Scanner Phase, Code Generation Phase, Testing codes on MIPS Simulator and report diagrams.