|  |
| --- |
| ARIZONA STATE UNIVERSITY |
| SHAW: REFERENCE MANUAL  SER 502, PROF. JOHN FEMIANI  By:  Jayanth Reddy Bogasamudram - 1209270448  Nithya Kogaleru - 1209379935  Meng Jung Lin - 1209373565 |

TABLE OF CONTENTS

[1. Introduction 3](#_Toc384832550)

[2. Compiler and run time environment 4](#_Toc384832551)

2.1 Compiler………………………………………………………………………………………………………..4

2.2 Runtime…………………………………………………………………………………………………………………….4

[3. Shaw type systems 6](#_Toc384832552)

[4. Examples 6](#_Toc384832553)

*4.1 Grammar………………………………………………………………………………………………………..6*

*4.2 Sample programs and parse trees……………………………………………………………………………7*

*4.3 Declaration Statements (int, bool, string)…………………………………………………………………..7*

*4.4 Conditional Statements (If - Else clause)…………………………………………………………………..8*

*4.5 Scope Definitions (Function declarations, while statement, recursion*)…………………………..10

[5. References 4](#_Toc384832554)

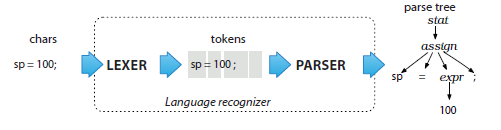
1. **INTRODUCTION**

The designed programming language has the name “Shaw”. The language design is inspired from Python and Java wherein the code structure resembles Java and the syntax of having no semi colons and no main function is taken up from Python. Further, the language acts as a “Shadow” to the two programming languages, thereby implementing its major functionalities and hence the name “Shaw”. The program represent the functionality and look of java but should incorporate the simplicity of python. Shaw is an abbreviation for Shadow.

Shaw is not just a programming language for programmers. It is also a programming language for people who learn compiler design. The programming language provides an option for a programmer to look at how a runtime is performing its operations.

This documentation is mainly useful for understanding the issues pertaining to the design as well as in implementing a new programming language. Following are the tools that have been used in the design and implementation of “Shaw”:

Tools and Prerequisites required:

* Antlr is a tool that has been implemented as a scanner/lexer and also for parsing.
* The compiler is written in java, so it is necessary to have JDK installed and the PATH is set up.
* Python 2.7 or older versions needs to be installed for run time.
* The tool is used to convert grammars into programs that recognizes sentences in the language described by the grammar.
*  Antlr tool generates recursive descent parser i.e. it implements top-down parser.

All the source files in Shaw have an extension of filename**.Shaw** extensions. The language Shaw is compiled and also interpreted just like any other programming language. The main purpose of Shaw compiler is to take up the source code provided and convert it into an intermediate code which is in turn fed into the run time environment that parses the given intermediate code line by line and finally executes it.

The Shaw complier is written in Java whereas the run time is written in Python.

1. **COMPILER AND RUN TIME ENVIRONMENT:**

**2.1 COMPILER:** The compiler has been implemented in java and the generated intermediate code has a **.txt** extensions. As antlr tool has been used to generate parse trees by using grammar which in turn generates java files. The grammar has .g4 extension and java files generated are as follows

* BaseVisitor.java
* Lexer.java
* Parser.java
* Visitor.java

This also generates two token files that contain individualized tokens. When Shaw is executed through antlr, the same classes are generated with prefix Shaw.

The BaseVisitor class contain all the nonterminal nodes that are written in grammar. MyVisitor class is used to override the BaseVisitor class in order to generate intermediate code by visiting each and every node generated by the parse tree.

* 1. **RUN TIME:** The run time is designed and implemented in Python. A set of opcodes were designed that the program will support, and then implement the runtime.

The run time works by executing the intermediate code line by line. The program will first read the source code and store it as an array of lines. The program will than go through the array one at time and perform appropriate operations.

Each op code has its own function, in Python we use hasattr to check whether an op code is valid, and getattr to call the function. Most of the functions in the runtime are straight forward and associate with the op codes. There is one supporting function find\_block, which is mainly used to find the location of a SBLOCK or EBLOCK.

In order for the runtime to work properly and as expected, we keep track of some variables within the code.

|  |  |  |
| --- | --- | --- |
| VARIABLES | TYPE | DESCRIPTION |
| declare\_func | boolean | Indicated whether the current status is declaring the function or actually running the function |
| block\_start | integer | The line number of SBLOCK when encountering one |
| block\_end | integer | The line number of EBLOCK when entering a block |
| run\_block | boolean | Indicate whether to run the immediate block of the next block, mainly used with op code CHECK |
| function\_dict | dictionary | Keeps track of the declared functions and their locations |

The interpreter was later developed using java. In order for the interpreter to work with the runtime, we have a small piece of code that will run python script in java.

The following is a table of the OPCODES and their description:

|  |  |
| --- | --- |
| OP CODES | DESCRIPTION |
| SR1 | Set R1 (Register 1) |
| SR2 | Set R2 (Register 2) |
| SR3 | Set R3 (Register 3) |
| PUSH | Push R1 to stack |
| SBLOCK | Beginning of a block |
| EBLOCK | Ending of a block |
| GET *OFFSET* | Set R1 = stack[*OFFSET*] |
| SET *OFFEST* | Set stack[*OFFSET*] = R1 |
| FNS *NAME* | Beginning of a function with *NAME* |
| FNE | Ending of a function |
| CALL *NAME* | Call the function |
| SW | Swap the values in R1 & R2 |
| JMP *LOCATION* | Jump to *LOCATION*. |
| JNE *LOCATION* | Jump to *LOCATION* if R1 != R2 |
| ADD | Set R1 = R1 + R2 |
| SUB | Set R1 = R1 - R2 |
| MUL | Set R1 = R1 \* R2 |
| DIV | Set R1 = R1 / R2 |
| CHECK | Compare the first two elements in the stack |
| PRT | Print R1 |
| EOF | End Of File |

Here LOCATION will be either SBLOCK or EBLOCK.

The following are the list of OPERATORS and their description:

|  |  |
| --- | --- |
| OP CODE | OPERATOR |
| LT | < |
| GT | > |
| LTEQ | <= |
| GTEQ | >= |
| EQ | == |
| NEQ | != |

1. **SHAW TYPE SYSTEMS**

Shaw is a statically-typed language, which implies that all the variables must first be declared before they can be used.

For example: int a = 5

The above statement indicates that the field name “a” exists, holds a numerical data and has an initial value of 5. A variable's data type determines the values it may contain, plus the operations that may be performed on it. In addition to int, Shaw supports two other data types which are as described below.

The data types supported by Shaw are as follows:

* int: Used for signed integer types.
* boolean: Used for declaring either true or false to a variable. It is a binary flag and can have only one of the two values.
* string: Implemented as an array of bytes that stores a sequence of characters.

1. **EXAMPLES**

All the Shaw source programs should end with a **.Shaw** extension while the intermediate file generated has a .txt extension.

4.1 **GRAMMAR:** The following is a snippet of grammar for Shaw. This grammar is used for declarations, expression evaluations and scope definitions. The grammar has .g4 extension.

stmt: block

| expression

| 'return' expression?

| 'while' parexpression stmt

|'if' parexpression stmt ('else' stmt)?

;

parexpression: '(' expression ')'

;

expression: expression( '==' | '!=' ) expression

| expression( '+' | '-' | '\*' | '/' ) expression

| expression ('<' | '<=' | '>' | '>=' | '=' ) expression

| '(' expression ')'

| expression '(' expression ')'

| Numval

| boolval

| Stringval

| Identifier

;

type: 'int'

| 'bool'

| 'String'

;

4.2 **SAMPLE PROGRAMS AND PARSE TREES:** Below are the sample programs and their respective parse trees.

4.3 **Declaration statements:** The following sample program and parse tree shows how Shaw accepts **“int”, “bool”, “String”** as primitive data types.

**Program:**

*[*

*{*

*int a = 10*

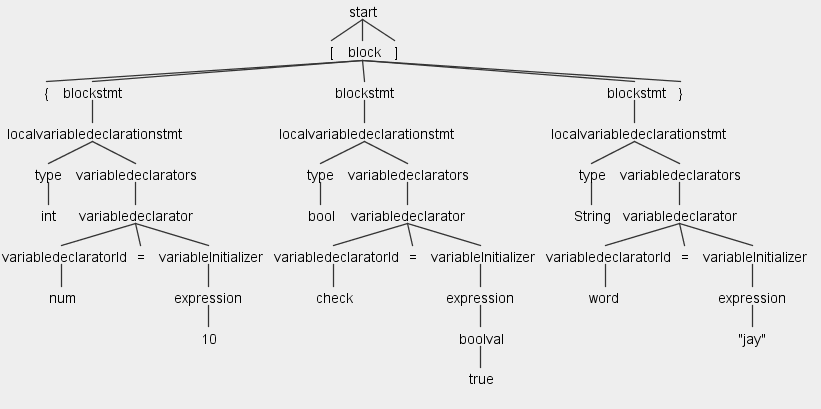
*bool c = true*

*String s = "jay"*

*}*

*]*

**Parse tree:**



**Intermediate Code:**

*SPROG*

*SR1 10*

*PUSH*

*SR2 TRUE*

*PUSH*

*SR3 JAY*

*PUSH*

*EPROG*

**Output:**

*10*

*true*

*jay*

4.4 **Conditional Statements:** The following sample program and parse tree shows how Shaw accepts **“If – Else”** as conditional statements.

**Program:**

*[*

*{*

*int a = 10*

*if (a == 10)*

*{*

*int b = 10*

*}*

*else*

*{*

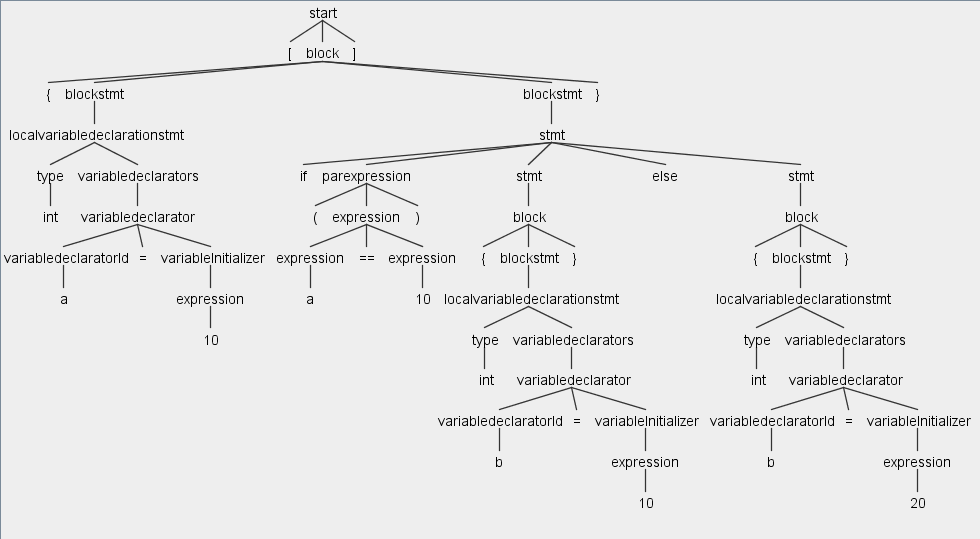
*int b = 20*

*}*

*}*

*]*

**Parse tree:**

****

**Intermediate Code:**

*SPROG*

*SR1 10*

*PUSH*

*SR1 10*

*PUSH*

*CHECK*

*SBLOCK*

*SR2 10*

*EBLOCK*

*SBLOCK*

*SR2 20*

*EBLOCK*

*EPROG*

**Output:**

No print statement is given for the program. So, if the conditions are satisfied, the compiler executes the program with no errors.

**4.5 Scope definitions:** The following sample program and parse tree shows how Shaw accepts **“function declarations”, “while statement” and “recursion”**.

**Program:**

*[*

*{*

*int a = 10;*

*int result = 1;*

*func int fact(int a)*

*{*

*while(a>1)*

*{ result = result \* a;*

*a = a-1;*

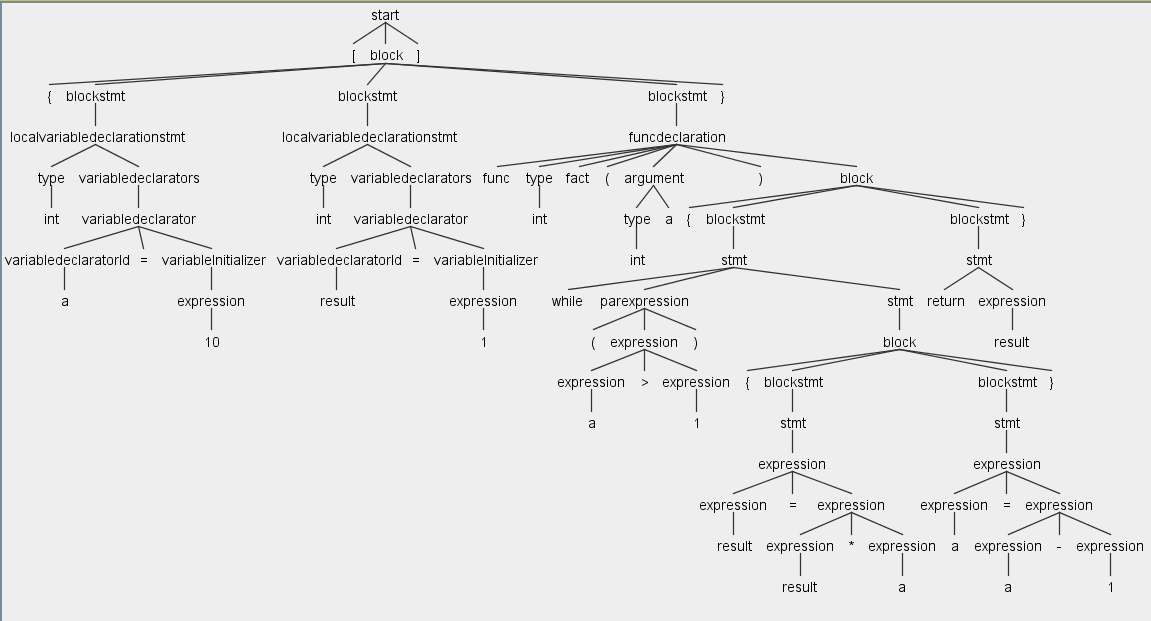
*}*

*return result;*

*}*

*}*

*]*

**Parse tree:**

**Intermediate Code:**

*SPROG*

*FNS fact*

*PUSH*

*PUSH*

*SR1 1*

*SET 1*

*SBLOCK*

*SR2 1*

*GET 0*

*GT*

*JNE EBLOCK*

*GET 1*

*SW*

*GET 0*

*MUL*

*SET 1*

*GET 0*

*SR2 1*

*SUB*

*SET 0*

*JMP SBLOCK*

*EBLOCK*

*GET 1*

*FNE*

*FNS test*

*SR1 5000*

*FNE*

*SR1 10*

*CALL fact*

*EPROG*

**Output:**

*3628800*

1. **REFERENCES:**

* [www.youtube.com](http://www.youtube.com)
* [www.google.com](http://www.google.com)
* [www.antlr.org](http://www.antlr.org)
* <https://pragprog.com/book/tpantlr2/the-definitive-antlr-4-reference>
* <http://stackoverflow.com>