LAKY - 502 Project Final Presentation

—

Team 33

Meet The Team

- Aaditya Muley
- Li Che Hsu
- Yunhan Qiao
- Karan Javali

The name LAKY was created by combining all of our initials and is pronounced as "LUCKY."

Language Description

1. It supports variable declarations with optional initial values for integers, strings, and booleans.
2. It has basic arithmetic expressions using addition, subtraction, multiplication, and division.
3. It has boolean expressions with relational operators, such as greater than, less than, equal to, and logical operators such as NOT, AND, and OR.
4. It supports simple conditional statements with an "if" clause and an optional "else" clause.
5. It has a ternary operator for conditional expressions.
6. It supports "while" loops.
7. It supports "for" loops with two different syntaxes for Java and Python style.
8. It supports basic input/output functionality through a "print" statement.

Syntax

Assignment:
int x := 5
bool y := true
string z := “I hope you enjoy working with LAKY”

Ternary operator:
tern (condition) ? (execute if true) : (execute if false) endtern

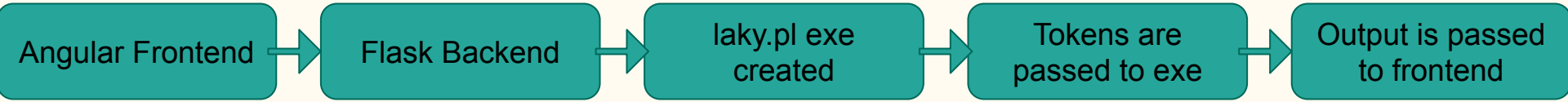
If-else:
if (condition) then (execute if true) else (execute if false) endif

For loop java-type:
for (variable assignment), (variable range) { (expressions to execute) } endforjava

For loop python-type:
for (variable name) inrange (range start), (range end) { (expressions to execute) } endforpython

While loop:
while (condition) do (expressions to execute) endwhile

Flowchart


Angular Frontend
Flask Backend
laky.pl exe created
Tokens are passed to exe
Output is passed to frontend

Grammar

<program> ::= <block> '.'

<block> ::= 'begin' <declare> 'end'

| 'begin' <declare> ';' <command> 'end'

<declare> ::= <declare1> ';'

| <declare1> ';' <declare>

<declare1> ::= 'int' <variable>

| 'int' <variable> ':=' <digit>

| 'string' <variable>

| 'string' <variable> ':=' <string>

| 'bool' <variable>

| 'bool' <variable> ':=' <boolean>

<command> ::= <command1> ';'

| <command1> ';' <command>

<command1> ::= <variable> ':=' <expression>

| 'if' <boolean> 'then' <command> 'else' <command> 'endif'

| 'tern' <boolean> '?' <command> ':' <command> 'endtern'

| 'for' <command1> <boolean> '{' <command> '}' 'endforjava'

| 'for' <variable> 'inrange' <digit> <digit> '{' <command> '}' 'endforpython'

| 'while' <boolean> 'do' <command> 'endwhile'

| 'print' <expression>

| <block>

<boolean> ::= 'true'

| 'false'

| <expression> <relational> <expression>

| 'not' <boolean>

<expression> ::= <temp1>

| <expression> '+' <temp1>

| <expression> '-' <temp1>

<temp1> ::= <temp2>

| <temp1> '*' <temp2>

| <temp1> '/' <temp2>

<temp2> ::= '(' <expression> ')'

| <variable> ':=' <expression>

| <variable>

| <digit>

<string> ::= '"' <string> '"'

<variable> ::= <identifier>

<digit> ::= <integer>

<relational> ::= '>' | '>=' | '<' | '<=' | ':=' | '!=' | '!' | '&&' | '||'

<identifier> ::= <alphabetic> <identifier> | <alphabetic>

<alphabetic> ::= 'a' | 'b' | 'c' | ... | 'y' | 'z' | 'A' | 'B' | 'C' | ... | 'Y' | 'Z'

<integer> ::= <digit> <integer> | <digit>

Parse Tree Generator

(Intermediate Code Generation)

SWIPL

As we described in our Milestone 1 submission, we were first torn between using ANTLR as an interpreter and sticking to what we already knew, namely SWIPL. However, due to time constraints (especially in light of other subjects' work), we decided to abandon the idea of learning ANTLR and instead use SWIPL to generate the intermediate code (parse-tree), which is then used by the evaluator to execute commands.

Overall, the combination of Prolog's logical and declarative nature, as well as SWIPL's features and capabilities, makes it an excellent choice for developing our language, with the biggest advantage being our familiarity with SWIPL.

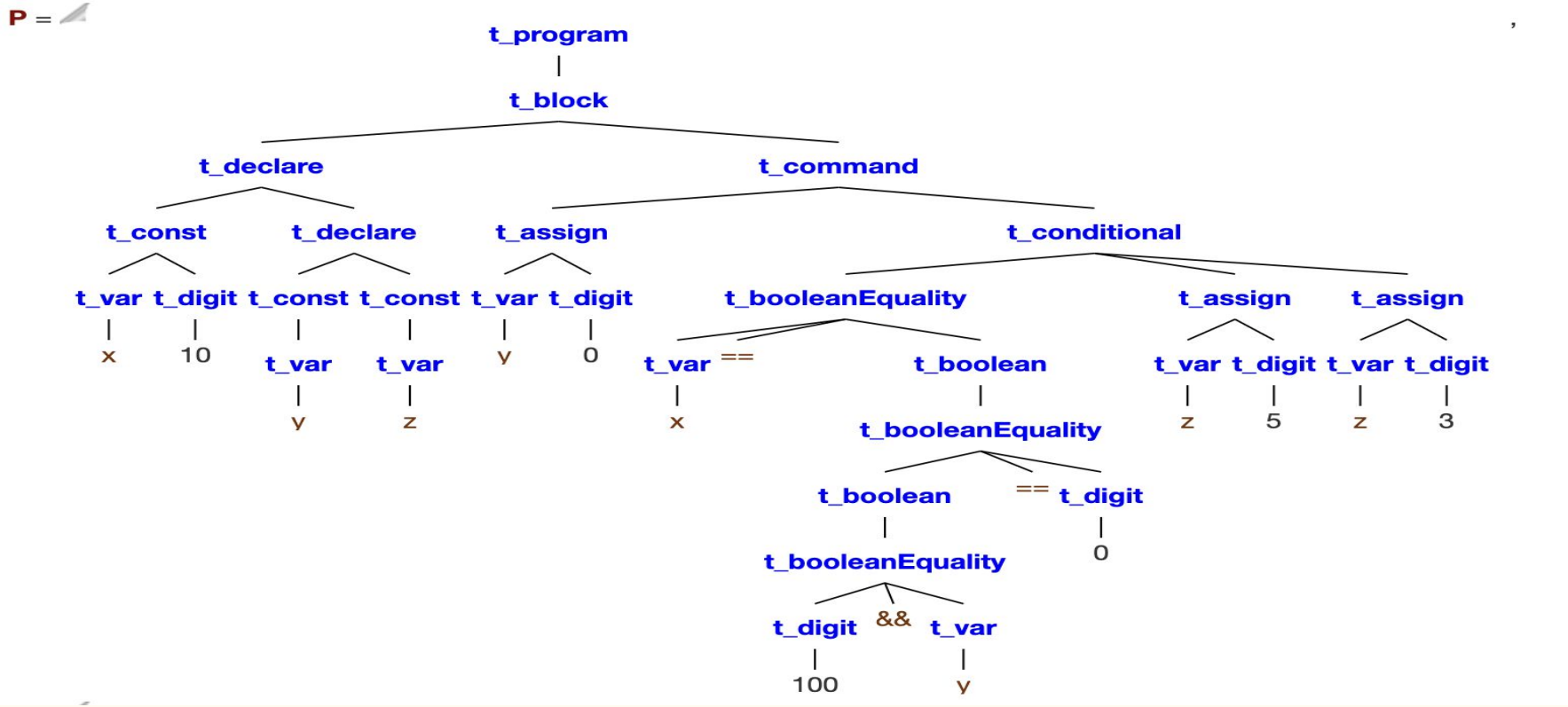
Sample program

```
begin
int x := 10;
int y; int z;
y := 0;
if x == 100 && y == 0
then z := 5
else z := 3
endif;
print z
end.
```

Intermediate Code

```
P
t_program(
t_block(
t_declare(
t_const(t_var(x),t_digit(10)),
t_declare(t_const(t_var(y)),t_const(t_var(z)))
)
,
t_command(
t_assign(t_var(y),t_digit(0)),
t_conditional(
t_booleanEquality(
t_boolean(
t_booleanEquality(
t_boolean(t_booleanEquality(t_var(x),==,t_digit(100))
),
=
&&,
t_var(y)
)
)
,
==,
t_digit(0)
)
,
t_assign(t_var(z),t_digit(5)),
t_assign(t_var(z),t_digit(3))
)
)
)
)
```

Parse Tree Visualization



Evaluator
(Runtime)

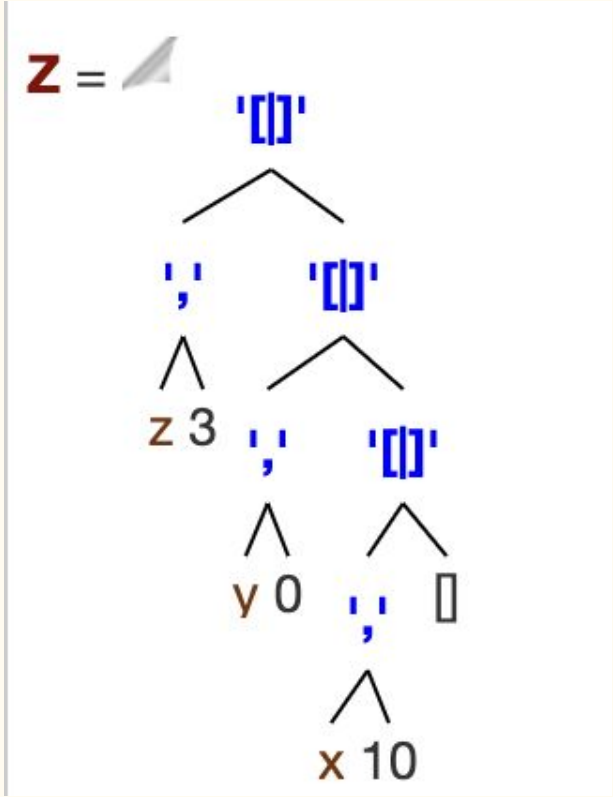
SWIPL

The evaluator, which is likewise written in SWIPL, takes the parser's intermediate code as input and evaluates it to actually execute all of the instructions.

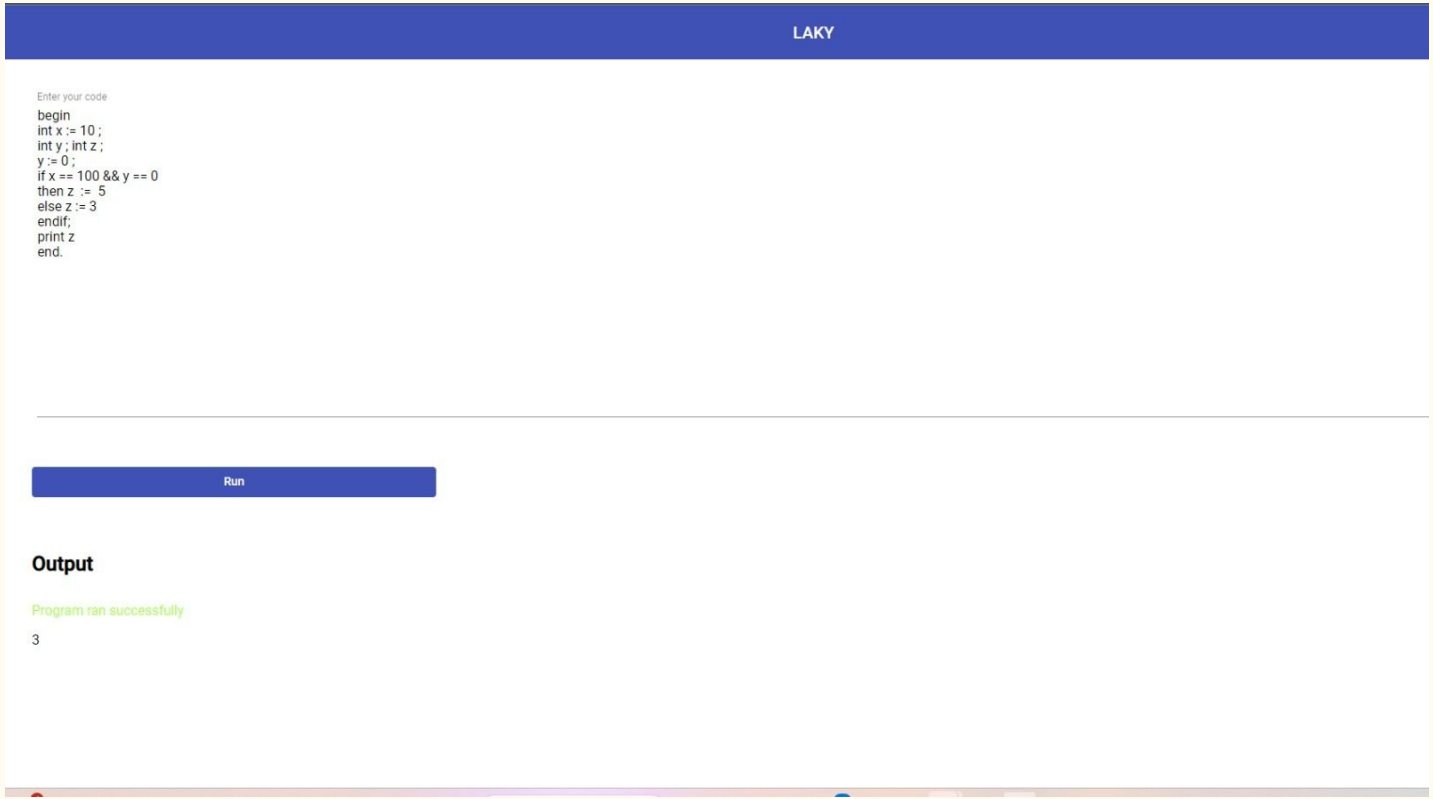
In the following slide, we show the output produced for the previously shown sample program.

This output is directly shown on our UI as output.

Output



Output


LAKY
Enter your code
begin
int x := 10 ;
int y ; int z ;
y := 0 ;
if x == 100 && y == 0
then z := 5
else z := 3
endif;
print z
end.
Run
Output
Program ran successfully
3