NO.

DATE / /

CC Project Handwritten Deliverables

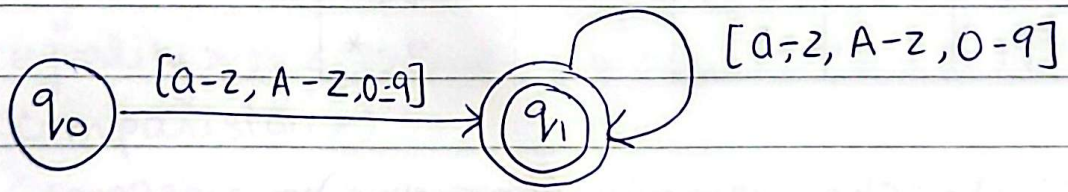
Group Members: Haneesh Ali, Waniya Syed, Valiharan Jalees

1. Lexical Analysis Artifacts

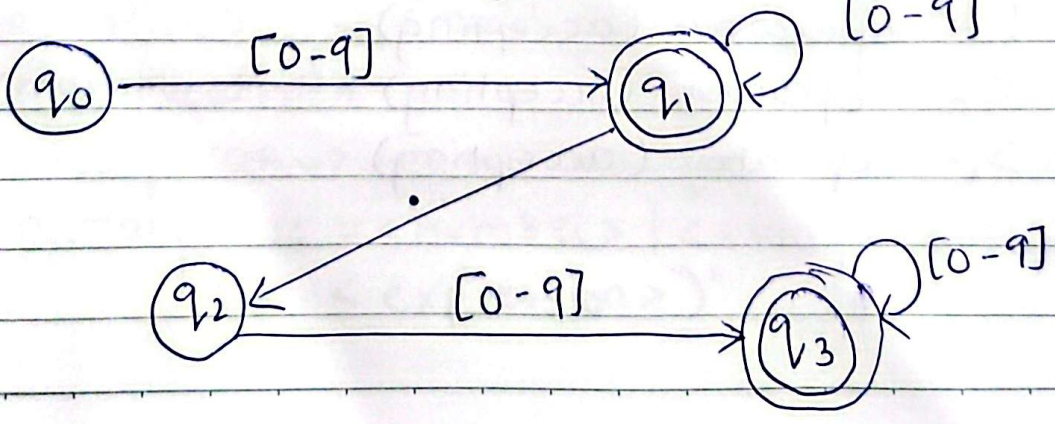
Token Definition Table:

Token Type	Examples
keyword	var, if, else, while, for, print
identifier	x, sum, counter, temp
number	42, 3.14, 0, 100
String	"Hello", "Done", "Result"
Operators:	+, -, *, /, =, ==, !=, <, >, <=, >=
seperator	( ) { } ; ,
comment	# This is a comment

DFA for identifier Recognition



DFA for number recognition (int & float)



NO.
DATE / /

Regular Expressions:

Keyword: var|if|else|while|for|print|function|return

identifier: [a-z,A-Z][a-z,A-Z,0-9]*

number: [0-9]+(\.[0-9]+)?

string: "([^"])*"

Operator: [+\-*/=!<>]=?|&&|\|\|

seperator: [(){};,]

comment: #.*

whitespace: [\t\n\r]+

Transition table for operators

State	=	!	<	>	other
q_0	q_1	q_2	q_3	q_4	q_5
q_1	q_6				q_5*
q_2	q_7				q_5*
q_3	q_8				q_5*
q_4	q_9				q_5*

q_5* : Single char operator (accepting)

q_6 : == operator (accepting)

q_7 : != operator (accepting)

q_8 : <= operator (accepting)

q_9 : >= operator (accepting)

NO.

DATE / /

2. Syntax Analysis Artifacts

Grammar Rules:

<program> ::= <statement>*

<statement> ::= <var_decl> | <assignment> |
<while_loop> | <if-statement> |
<print_statement>

<var_decl> ::= "var" <identifier> "=" <expression>

<assignment> ::= <identifier> "=" <expression> ")" ";"

<while_loop> ::= "while" "(" <expression> ")" "{"
<statement>* "}"

<if_statement> ::= "if" "(" <expression> ")" "{"
<statement>* "}"

<print_statement> ::= "print" "(" <expression> ")" ";"

<expression> ::= <logical_or>

<logical_or> ::= <logical_and> ("||" <logical_and>)*

<logical-and> ::= <equality> ("&&" <equality>)*

<equality> ::= <comparision> (("==" | "!=")
<comparision>)*

<comparision> ::= <additive> (("<" | ">" | "<=" | ">=")
<additive>)*

<additive> ::= <multiplicative> (("+" | "-")
<multiplicative>)*

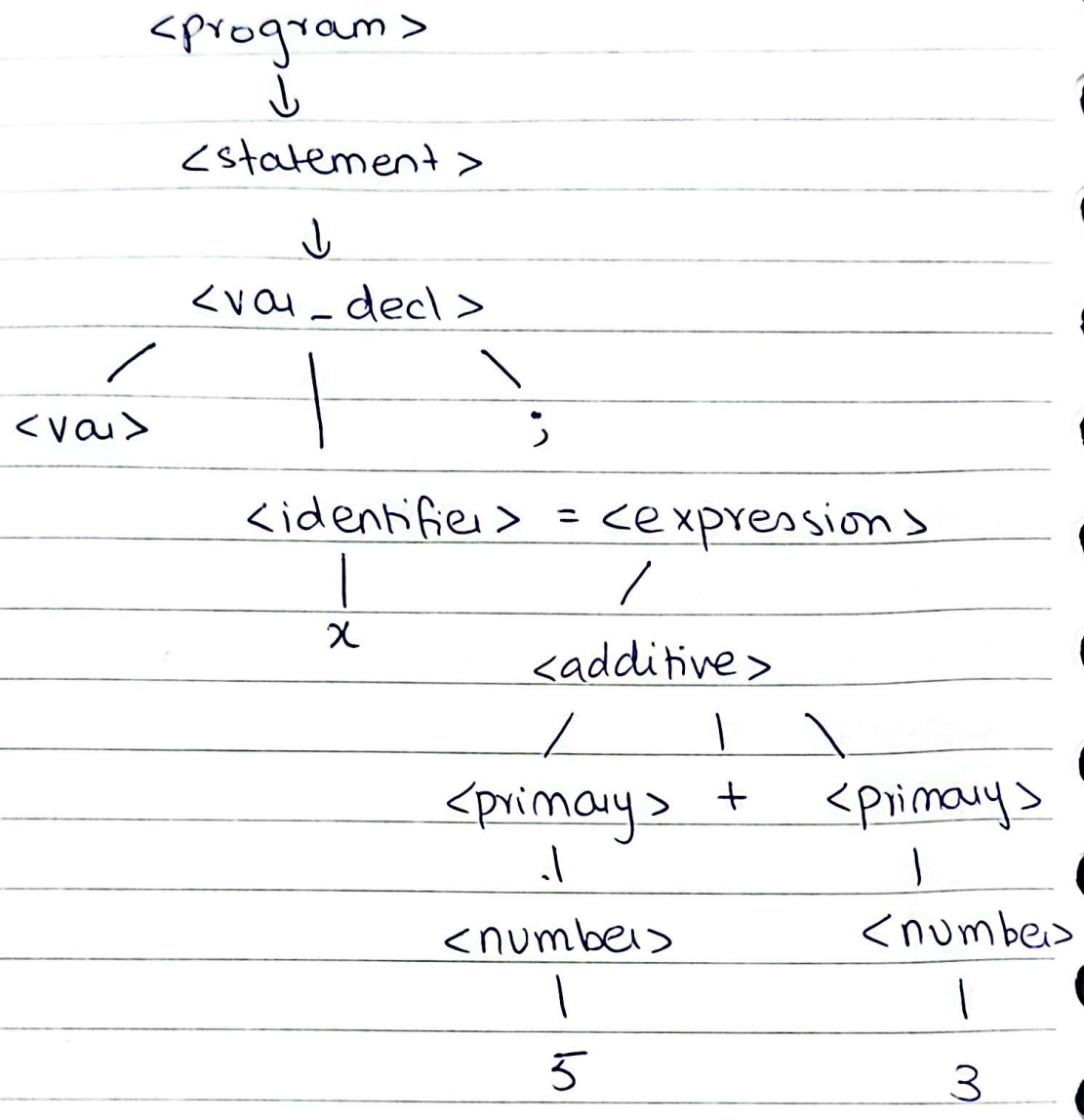
<multiplicative> ::= <primary> (("*" | "/") <primary>)*

<primary> ::= <number> | <string> | <identifier> | "("
<expression> ")"

NO.
DATE / /

Parse Tree Example 1:

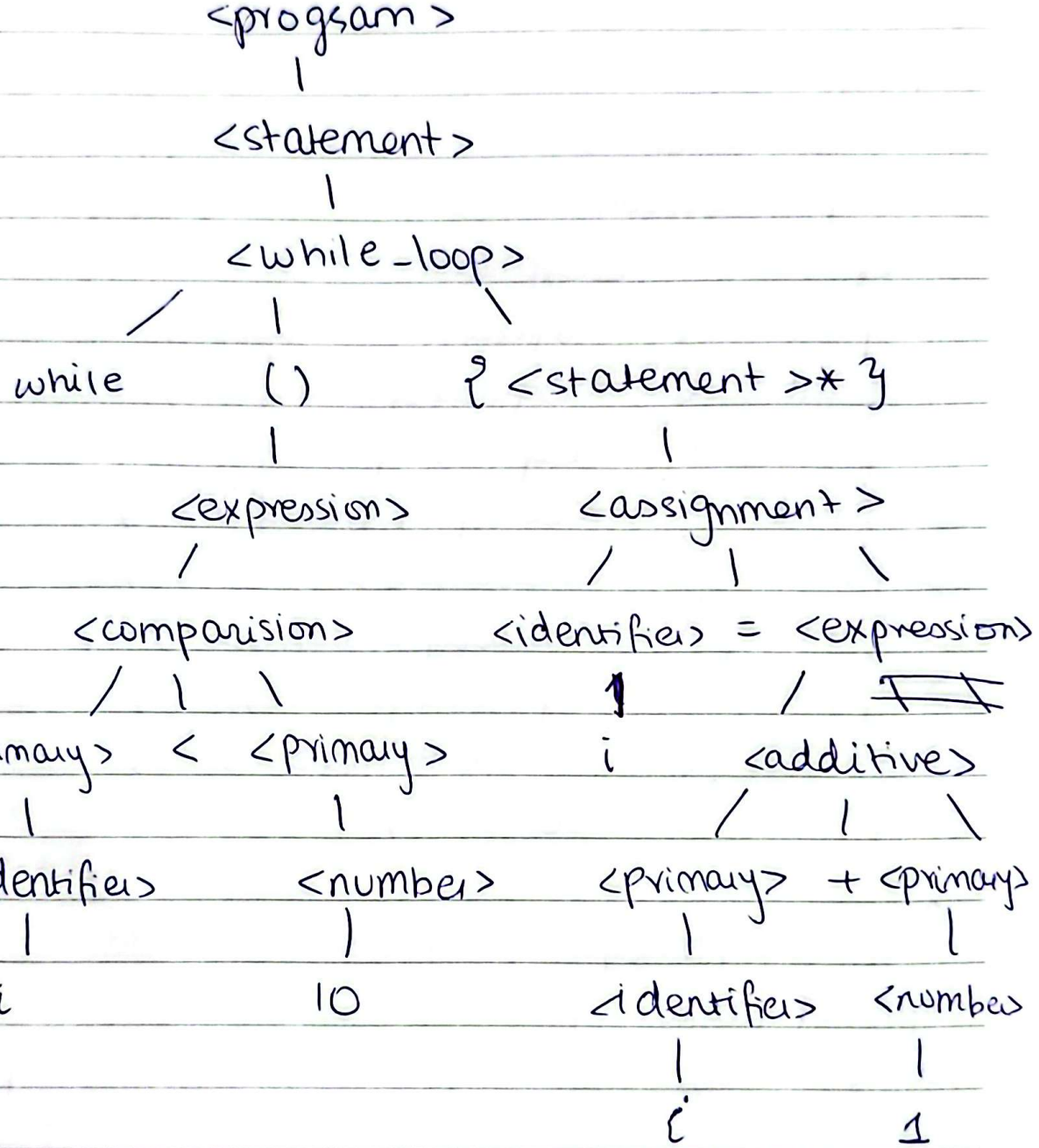
var x = 5 + 3;



NO

DATE / /

Pause Tree example 2: while (i<10){i = i+1;}



NO.
DATE / /

Leftmost Derivation for print (a):

<program>
=> <statement>
=> <print_statement>
=> "print" "(" <expression> ")" ";"
=> "print" "(" <logical_or> ")" ";"
=> "print" "(" <logical_and> ")" ";"
=> "print" "(" <equality> ")" ";"
=> "print" "(" <comparision> ")" ";"
=> "print" "(" <additive> ")" ";"
=> "print" "(" <multiplicative> ")" ";"
=> "print" "(" <primary> ")" ";"
=> "print" "(" <identifier> ")" ";"
=> "print" "(" a ")" ";"

3. Semantic Analysis Artifacts

Symbol table for:

var x = 10; var y = 20; x = x + y;

Name	Type	Scope	Line	Status
x	variable	global	1	declared
y	variable	global	2	declared

NO.

DATE / /

Symbol Table with Scope

code: var x = 5

```
while (x<10) {
    var y = x * 2;
    x = x + 1; }
var y = 100;
```

Name	Type	Scope	Line	Status
x	variable	global	1	declared
y	variable	whileloop	3	declared
y	variable	global	5	declared

Type Checking Example

```
var x = 10;
var y = 20;
var z = x+y;
var msg = "Hello";
var bad = x + msg;
```

Expression	Type	Result	Status
10	Number	Number	✓
x+y	Number	Number	✓
"Hello"	String	String	✓
x+msg	mixed	error	x

11.11 Date:

Intermediate Code:

Implementation Strategy- The Intermediate code generator traverses the Abstract Syntax Tree (AST). For every operation, it creates a new temporary variable (t_n) to store the result. Arithmetic operations are broken down into simple [temp= left op right] instructions. Control flow is managed using goto statements and unique labels(L-n). Conditionals evaluate to a temporary variable; if the condition is false, the program jumps to a specific label to skip code blocks.

Example 1: Arithmetic Expression

Source code(MEL):

```
var result = (10+5) ^ 2
```

Generated Three Address code:

```
t0 = 10 + 5
t1 = t0 ^ 2
result = t1
```

Example 2: If-Else Statement

Source code(MEL):

```
if (x>10) {
    y=1;
} else {
    y=0;
}
```

Generated Three Address code:

```
t0 = x > 10
if t0 == 0 goto L0
y = 1
goto L1
L0:
    y = 0
L1:
```

Date:

Example 3: While Loop

source code(MEL):

```
while(i<n){
    i=i+1;
}
```

Generated Three Address code:

```
L0:
    to = i < n
    if to ==0 goto L1
    t1 = i+1
    i = t1
    goto L0
L1:
```

Label Generation Logic

The compiler maintains a global counter for labels to ensure uniqueness

1) For While Loops: Two labels are generated. The start label marks the beginning of the condition check to allow looping. The end label is the target for the jump instruction when the condition evaluates to false, allowing the program to exit the loop.

2) For If-Else: Two labels are generated. The ~~start~~ else label marks the start of the else block. The end label marks the end of the entire structure so the "True" block can skip the "Else" block after Execution.

Date:

Optimization

optimization techniques

1) **Constant Folding:** The optimizer scans the Three Address Code for operations involving two immediate numbers (e.g 5+3). It calculates the result at compile time and replaces the operation with the calculated result directly in the code.

2) **Constant Propagation:** Once a variable or temporary is assigned a constant value, that value is substituted into subsequent instructions that use the variable, often enabling further folding.

optimization Example: Constant Calculation

source code (MEL):

```
var x = (10 * 5) + 2;
```

Intermediate code (Before optimization)	Optimized code
$t0 = 10 * 5$	$t0 = 50$
$t1 = t0 + 2$	$t1 = 52$
$x = t1$	$x = 52$

Step by Step optimization process

1- **Line 1:** The optimizer identifies [10 * 5]. Since both operands are numbers, it computes 50 and replaces the line with t0=50.

2- **Line 2:** The code was t1 = t0 + 2. The optimizer sees t0 is now a known constant (50). It propagates the value to create 50 + 2, which it then folds to 52.

3- **Line 3:** The code was x = t1, The optimizer knows $t_1 = 52$, so it updates the assignment to x = 52.

Date:

Reflection.

What we Learned

- **Pipeline Design:** Mastered chaining 6 distinct phases where output from one (e.g Tokens) strictly feeds into the next (e.g AST)
- **Recursive Descent:** Gained deep understanding of converting BNF grammar into recursive Python functions to parse complex structures.
- **Symbol Management:** Learned to track variable scope and types to prevent logic errors before execution

Challenges Faced

- **Operator Precedence:** Enforcing multiplication before addition was solved by stratifying the parser into -expression, -term, and -factor methods
- **Control Flow TAC:** Generating correct goto labels for nested If/Else blocks required precise label counter management to prevent infinite loops

Future Improvements

- **Functions:** Add FunctionDeclaration nodes and a CallStack to support recursion and reusable code.
- **Native Code:** Generate x86 Assembly or C instead of interpreting in Python for better performance