



# SER502 Project DevilsCode

Team 5



**Hari Iyer  
Pankaj Singh  
Ting Yeu Yang  
Mingfei Yang**

# Outlines

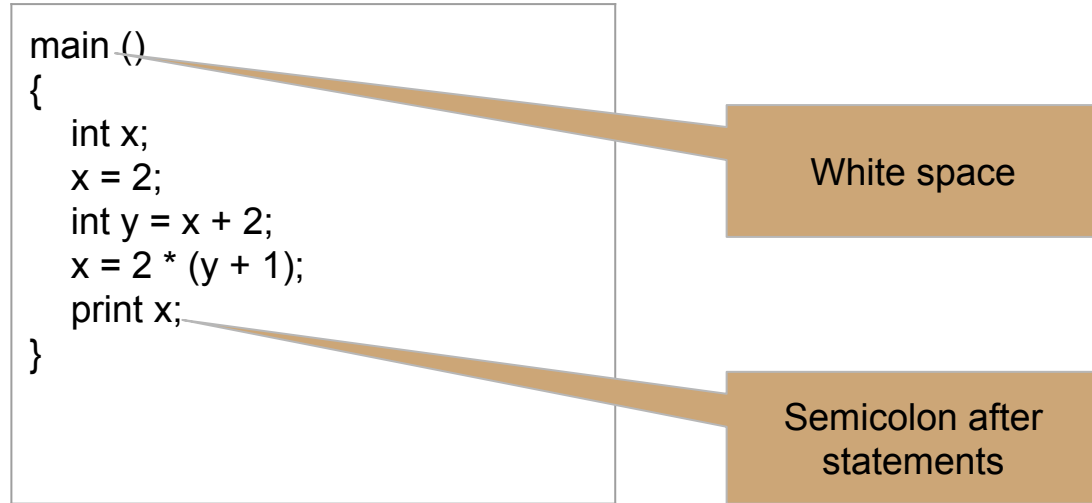
- Introduction
- Syntax & Grammar
- Lexical Analyzer & Parser
- Intermediate Code Generation
- Runtime

# Introductions

- Designed language name: DevilsCode
- System: Ubuntu 16.04 LTS
- Tools:
  - Compiler: Python 3.5.2 and ANTLR 4.7
  - Runtime: Java SE 1.6
- Basic features:
  - Supports data types: integer and boolean
  - Supports while-loop
  - Supports if-else-statement
  - An output function to show results on console
  - Extension: .dvlc for source code and .ic for intermediate code

# Syntax

- Code structure, declaration, assignment, and mathematical statements



# Syntax (cont')

- while-loop statement

```
while (count > 5)
{
    print count;
    count = count + 1;
}
```

Does not support  
multiple conditions

# Syntax (cont')

- if-else statement

```
if (age < 18)
{
    print 0;
}
```

```
if (age < 21)
{
    print 1;
}
```

```
else
{
    print 2;
}
```

Keyword “else” is optional

# Grammar

Original grammar rules



.g4 file for ANTLR input

## Statement

```
<stmt> ::= <expr> ';'
          | <dec_stmt> ';'
          | <while_stmt>
          | <if_stmt>
          | '{' <stmts> '}'
```

```
<stmts> ::= <stmts> <stmt> | ε
```

## Expression

```
<expr> ::= <id>
          | <assign_expr>
          | <math_expr>
          | <cpr_expr>
```

```
9 grammar DevilsCode;
10
11 /** Parser Rules */
12 /** Start Rule */
13 program
14     : 'main ' '(' ' ' ')'
15     '{'
16     stmtlists
17     '}'
18     ;
19
20 /** Statement Lists
21 * <stmts> ::= <stmts> <stmt> | ε
22 */
23 stmtlists
24     : stmtlists stmt
25     | /* epsilon */
26     ;
27
28 /** Statement
29 * <stmt> ::= <expr> ';'
30 *         | <dec_stmt> ';'
31 *         | <while_stmt>
32 *         | <if_stmt>
33 *         | '{' <stmts> '}'
34 */
35 stmt
36     : expr ';'
37     | dec_stmt ';'
38     | while_stmt
39     | if_stmt
40     | '{' stmtlists '}'
41     | 'print ' IDENT ';'
42     | 'print ' number ';'
43     ;
44
45 /** Expression
46 * <expr> ::= <id>
47 *         | <assign_expr>
48 *         | <math_expr>
49 *         | <cpr_expr>
50 */
51 expr
52     : IDENT
53     | assign_expr
54     | math_expr
55     | cpr_expr
56     ;
```

# Lexical Analysis

- ANTLR - Translates Grammar to Lexer and Parser
- DevilsCode Input
- Lexical Rules
- Keywords and Literals Validation
- Error Reporting

```
if __name__ == '__main__':  
    if len(sys.argv) > 1:  
        input_stream = FileStream(sys.argv[1])  
    else:  
        input_stream = InputStream(sys.stdin.readline())  
  
    file_name = sys.argv[1].rsplit('.', 1)[0]  
    lexer = DevilsCodeLexer(input_stream)  
    token_stream = CommonTokenStream(lexer)  
    parser = DevilsCodeParser(token_stream, file_name)  
    tree = parser.program()  
  
    tree_str = tree.toStringTree(recog=parser)  
    print(tree_str)  
    listener = DevilsCodeListener(file_name)  
    walker = ParseTreeWalker()  
    walker.walk(listener, tree)  
    listener.symbolTableChecker()
```



# Parsing

- Parser generated from the Rules
- Consumes TokenStream from Lexer
- Analyze Code for its Grammatical Structure
- program() as Entry Point to the Code
- Generates Parse Tree

```
hariyer@lufthansa454:~/devils/compiler$ cat compare.dvlc
main ()
{
    int a = 10;
    if (a < 15)
    {
        print 0;
    }
    else
    {
        print 1;
    }
}

hariyer@lufthansa454:~/devils/compiler$ dvlc compare.dvlc
(program main ( ) { stmtlists stmtlists (stmt (dec_stmt int (assign_expr a = (expr (math_expr (math_term (math_factor (number (number 1) 0)))))) ;)) (stmt (if_stmt if ( (expr (cpr_expr (cpr_term a) < (cpr_term (number (number 1) 5)))) ) (stmt { (stmtlists stmtlists (stmt print (number 0) ;)) } ) (else_stmt else (stmt { (stmtlists stmtlists (stmt print (number 1) ;)) })))))) ;))
hariyer@lufthansa454:~/devils/compiler$
```

# Parse Tree

- Syntactic Structure of the Program
- Precursor to Intermediate Code
- Listener enables Parse Tree Traversal
- getChild() and getParent() for depth-traversals
- Example:

**Code:**

```
main ()  
{  
    int i = 1;  
    i = i + 120;  
    print i;  
}
```

**Parse Tree:**

```
(program main ( ) { (stmtlists (stmtlists (stmtlists stmtlists  
(stmt (dec_stmt int (assign_expr i = (expr (math_expr  
(math_term (math_factor (number 1)))))) ;)) (stmt (expr  
(assign_expr i = (expr (math_expr (math_expr (math_term  
(math_factor i))) + (math_term (math_factor (number  
(number (number 1) 2) 0)))))) ;)) (stmt print i ;)) })
```



# Intermediate Code Generation



# Intermediate Code Format

- Similar to Assembly language
- Output Format: **Line Number** + **Keyword** (+ **Argument**)
- **Keywords:**
  - **START/STOP**: Mark the beginning and the end of source program
  - **BEGIN/END**: Mark the beginning and the end of statement block ( i.e. { stmt } )
  - **DEC**: Mark the variable declaration, takes two arguments
  - **ASSN**: Mark the assignment expression, takes two arguments
  - **ADD/SUB/MUL/DIV**: Mark the arithmetic operations takes two arguments
  - **SML/SMLEQL/GTR/GTREQ/L/EQL**: Mark the comparison operations, two args
  - **IF/ELSE**: Mark the beginning of certain condition statement
  - **LOOP**: Mark the beginning of iteration statement
  - **PRINT**: Mark the print statement, takes one argument
  - **GOTO**: Mark the branch target, takes one argument

# Generation Procedure

- Intermediate Code is generated by traversing parse tree
- Implemented in [DevilsCodeListener.py](#), which is used for traversal
  - For each Non-terminal NT in grammar rules, ANTLR will generate two functions inside Listener named “enterNT” and “exitNT”
  - Certain code is written into the file when “entering” or “exiting” one node
    - Entering: “enterNT” function is called when visiting this node
    - Exiting: “exitNT” function is called after traversal of all children of this node
- Enter functions are usually used to generate intermediate code with [START, BEGIN, DEC, SML/SMLEQL/GTR/GTREQL/EQL, PRINT, IF/ELSE, LOOP](#)
- Exit functions are usually used to generate intermediate code with [STOP, END, ADD/SUB/MUL/DIV, ASSN, GOTO](#)

# Data Structure Used in Generation Procedure

- Stack
  - Temporary values storage
  - Nested loops handling
  - Matching `else` with the nearest `if`
  - Branch target handling

# Sample

```
main ()
{
    int i;
    int f = 5;
    int a = 3;
    int b = 4;
    int c = 10;
    int d = 5;

    i = (10 * 13 + (a + b)) * ((c - d) * (15 + 5));

    if ( i < f )
    {
        i = i + 3 * 2;
    }
    else
    {
        i = i - 1;
    }

    while (f > 1)
    {
        f = f - 1;
    }

    print i;
    print f;
}
```

```
0 START
1 DEC i INT
2 DEC f INT
3 ASSN f 5
4 DEC a INT
5 ASSN a 3
6 DEC b INT
7 ASSN b 4
8 DEC c INT
9 ASSN c 10
10 DEC d INT
11 ASSN d 5
12 MUL 10 13
13 ADD a b
14 ADD TOP TOP
15 SUB c d
16 ADD 15 5
17 MUL TOP TOP
18 MUL TOP TOP
19 ASSN i TOP
20 IF
21 SML i f
22 GOTO 27
23 BEGIN
24 MUL 3 2
25 ADD i TOP
26 ASSN i TOP
27 END
28 ELSE
29 BEGIN
30 SUB i 1
31 ASSN i TOP
32 END
33 LOOP
34 GTR f 1
35 GOTO 40
36 BEGIN
37 SUB f 1
38 ASSN f TOP
39 GOTO 33
40 END
41 PRINT i
42 PRINT f
43 STOP
```

# DevilsCode Runtime

- Implemented in Java language
- Requires Java 1.6 or higher to support runtime
- Whole code is packaged a jar file and exported
- Command to execute runtime separately

```
java -jar DevilsCodeRuntime [intermediate code filename]
```



# DevilsCode Runtime

Reads intermediate code as a list of statements

- Reads intermediate code from file
- Stores the code in the form of List of statements
- Executes each statement one by one

# Statements

- Each statement has at max 4 components
  - Line Number : represent index in the list
  - Operation : a keyword with special meaning to runtime e.g. ASSN to assign values
  - Operand1 : variable or data value required by operation
  - Operand2 : variable or data value required by operation
- Variables are stored in a Map with name as key and its state as value
- Two separate Maps are used for integer and boolean

```
public class Statement {  
  
    private int lineNo;  
    private String operation;  
    private String op1;  
    private String op2;  
}
```

# Arithmetic and Boolean Operations

- All supported Arithmetic operations have 2 operands e.g.

ADD a b

Which means  $a+b$ ; where a and b are integer variables

- Stores the result in intStack with last value pushed at the TOP
- During assignment TOP is popped from stack and assigned to variables

ASSN c TOP

- Similar procedure is implemented for boolean operation

# Branching Operation

- IF and ELSE marks the branching statement
- IF will be followed by a condition statement
- Condition will be evaluated and if it is true we skip the next statement else continue
- Next Statement is GOTO line\_number ; it sets the index to a new lineNumber and the runtime jumps to start executing from there (i.e. END of IF block)
- For ELSE statement, runtime checks if the condition was false then skip GOTO statement otherwise continue

```
0 START
1 DEC a INT
2 ASSN a 10
3 IF
4 SML a 15
5 GOTO 8
6 BEGIN
7 PRINT 0
8 END
9 ELSE
10 BEGIN
11 PRINT 1
12 END
13 STOP
```

# LOOP Operation

- LOOP statement marks the looping block
- LOOP will be followed by a condition statement
- Condition will be evaluated and if it is true we skip the next statement else continue
- Next Statement is GOTO line\_number ; it sets the index to a new lineNumber and the runtime jumps to start executing from there (i.e. END of LOOP block)
- Before the end of Loop block there is another GOTO statement which makes runtime come back at the LOOP statement.

```
//branching statements
case "IF" : branching =true;
           break;
case "ELSE" : if(branching&&!gotoSkipped)
               index++;//skip GOTO statement
           break;
case "LOOP" :branching =true;
           break;
case "GOTO" ://implement GOTO
             int i=ariOps.executeGOTO(statements,smt);
             index=i-1;
             //branching =true;
             break;
```

```
private boolean skipGOTO(boolean branching)
{
    String stat=statements.get(index+1);
    boolean val=stat.contains("GOTO");
    if(val&&branching&&ReservedKeywords.getTopb())
    {
        index++;//skip GOTO statement
        return true;
    }
    return false;
}
```

# Execute Compiler and Runtime

