Compiler Theory

The 1st Program Assignment – Scanner Construction
21900628 – Sechang Jang

Index

Theoretical Foundations

BNF (Backus-Naur Form)

BNF is a notation technique used to express the grammar of a language in a formal way. It is widely used in the field of computer science to describe the syntax of programming languages, data structures, and document formats. BNF provides a set of derivation rules composed of a sequence of symbols, where each rule describes one of the constructs of the language. It uses ::= to denote definition, | to indicate alternatives, and terminal and non-terminal symbols to represent the basic elements and constructs of the language, respectively.

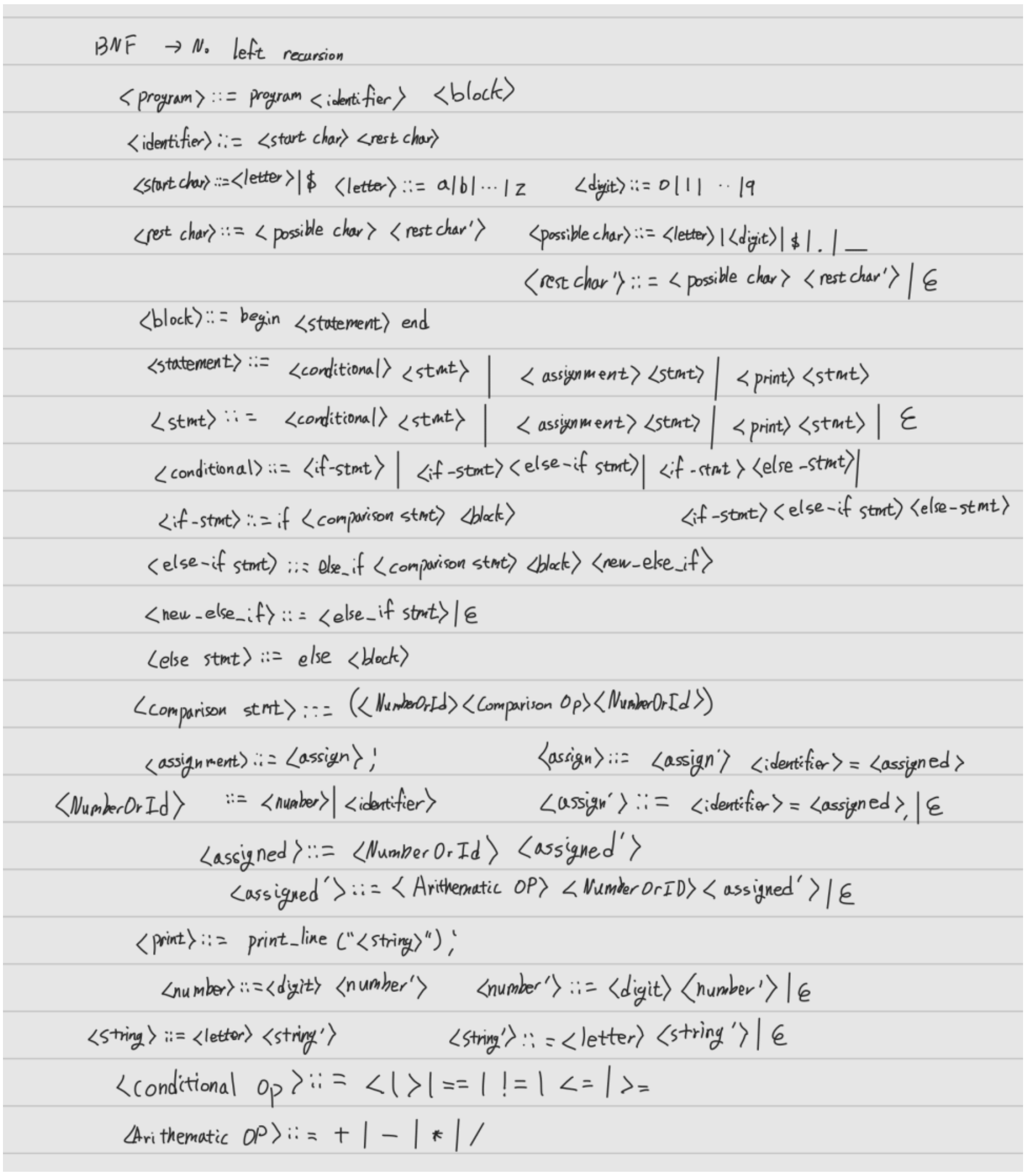
EBNF (Extended Backus-Naur Form)

EBNF is an enhancement of the original BNF, adding more expressive power through additional constructs. These include optional elements, repetitions, and groupings, which make EBNF more flexible and easier to use than standard BNF. This extended form allows for more concise and understandable grammar representations by incorporating symbols like [ ] for optional parts, { } for repetition, and ( ) for grouping, thereby reducing the complexity and length of grammar definitions compared to traditional BNF.

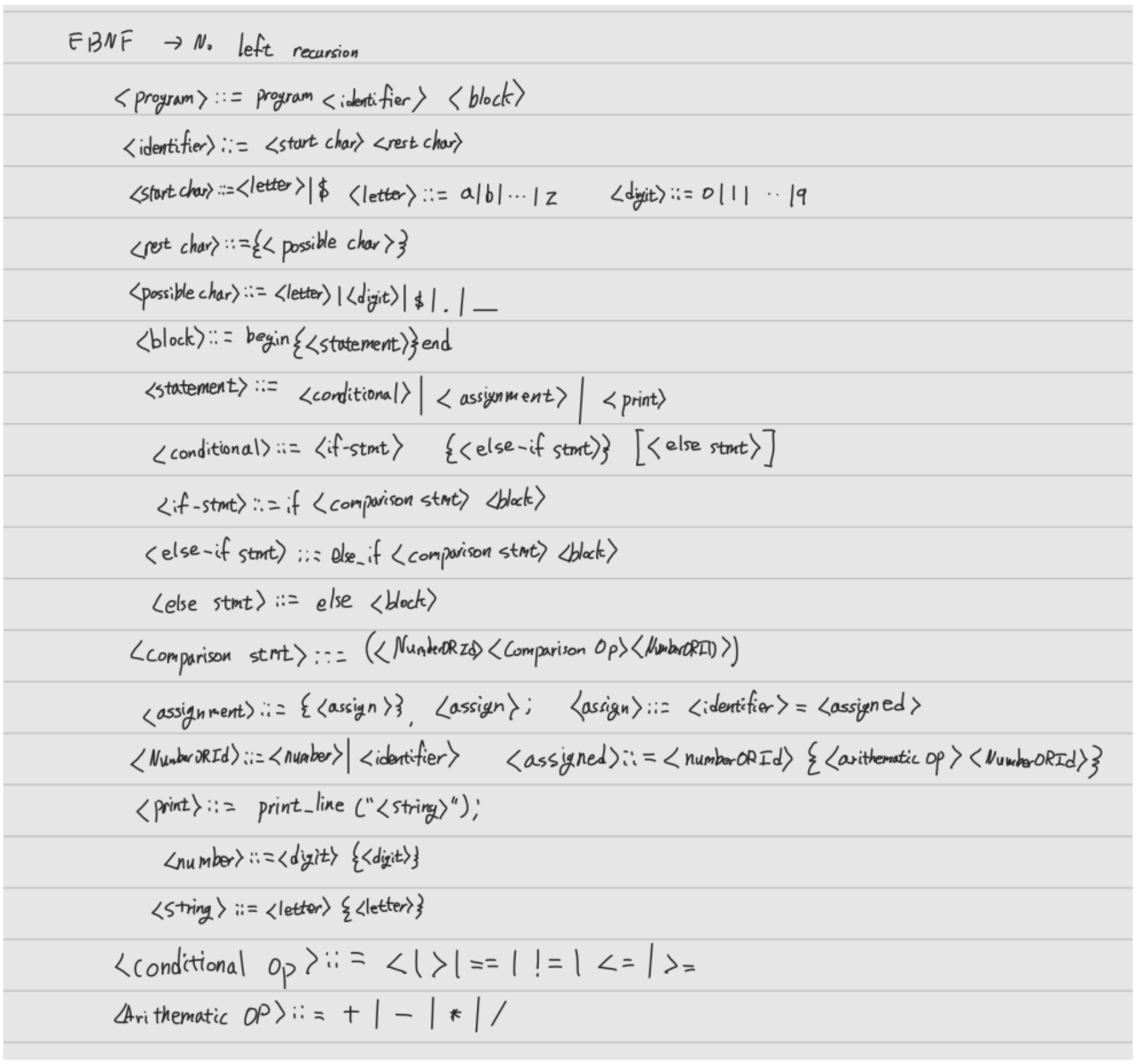
Left Recursion Elimination

Left recursion occurs in grammars when a non-terminal symbol in a production rule refers to itself as its first element. This can be problematic for certain types of parsers, like recursive descent parsers, because it can lead to infinite recursion. To handle this, left recursion needs to be eliminated from grammar rules. This is done by transforming recursive production rules into equivalent non-recursive ones, often by introducing new non-terminal symbols and rearranging the order of production rules to ensure that recursive calls are no longer made as the first action, thus allowing parsers to handle the grammar correctly without infinite loops. Left recursion elimination is a crucial step in preparing a grammar for many parsing techniques.

BNF and EBNF

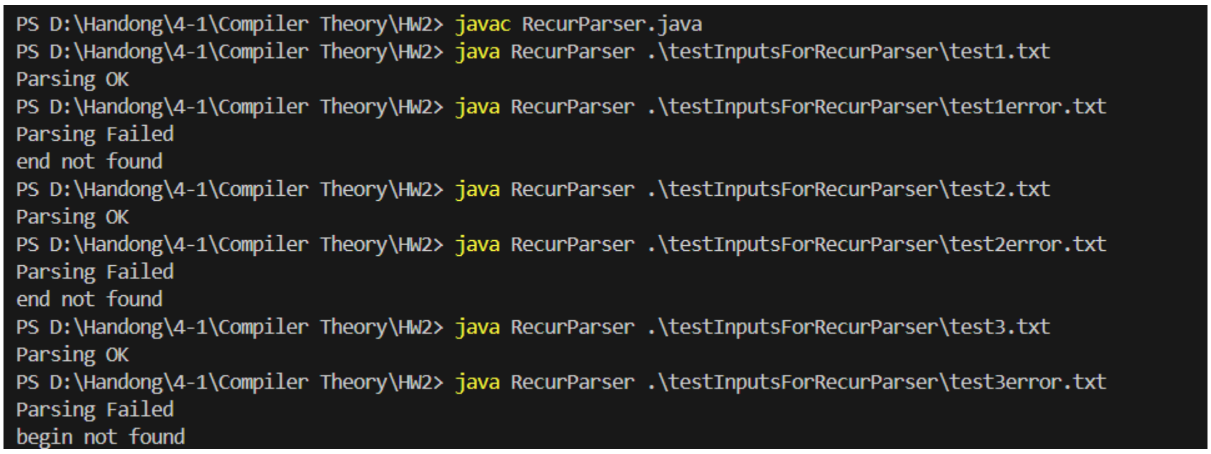


```
BNF → No left recursion

<program> ::= program <identifier> <block>
<identifier> ::= <start char> <rest char>
<start char> ::= <letter> | $     <letter> ::= a | b | ··· | z     <digit> ::= 0 | 1 | ·· | 9
<rest char> ::= <possible char> <rest char'>     <possible char> ::= <letter> | <digit> | $ | . | _
                                                 <rest char'> ::= <possible char> <rest char'> | ε
<block> ::= begin <statement> end
<statement> ::= <conditional> <stmt> | <assignment> <stmt> | <print> <stmt>
<stmt> ::= <conditional> <stmt> | <assignment> <stmt> | <print> <stmt> | ε
<conditional> ::= <if-stmt> | <if-stmt> <else-if stmt> | <if-stmt> <else-stmt> |
<if-stmt> ::= if <comparison stmt> <block>            <if-stmt> <else-if stmt> <else-stmt>
<else-if stmt> ::= else_if <comparison stmt> <block> <new_else_if>
<new_else_if> ::= <else_if stmt> | ε
<else stmt> ::= else <block>
<Comparison stmt> ::= (<NumberOrId> <Comparison Op> <NumberOrId>)
<assignment> ::= <assign> ;            <asign> ::= <assign'> <identifier> = <assigned>
<NumberOrId> ::= <number> | <identifier>       <assign'> ::= <identifier> = <assigned>, | ε
        <assigned> ::= <Number Or Id> <assigned'>
            <assigned'> ::= <Arithematic OP> <NumberOrID> <assigned'> | ε
<print> ::= print_line ("<string>");
<number> ::= <digit> <number'>       <number'> ::= <digit> <number'> | ε
<string> ::= <letter> <string'>      <string'> ::= <letter> <string'> | ε
<conditional op> ::= < | > | == | != | <= | >=
<Arithematic OP> ::= + | - | * | /
```



```
EBNF  → No left recursion

<program> ::= program <identifier> <block>
<identifier> ::= <start char> <rest char>
<start char> ::= <letter> | $   <letter> ::= a|b| ··· |z    <digit> ::= 0|1| ·· |9
<rest char> ::= {<possible char>}
<possible char> ::= <letter> | <digit> | $ | . | _
<block> ::= begin {<statement>} end
<statement> ::= <conditional> | <assignment> | <print>
<conditional> ::= <if-stmt>   {<else-if stmt>}  [<else stmt>]
<if-stmt> ::= if <comparison stmt> <block>
<else-if stmt> ::= else_if <comparison stmt> <block>
<else stmt> ::= else <block>
<Comparison stmt> ::= (<NumberORId> <Comparison Op> <NumberORID>)
<assignment> ::= {<assign>},  <assign>;   <assign> ::= <identifier> = <assigned>
<NumberORId> ::= <number> | <identifier>    <assigned> ::= <numberORId> {<arithematic op> <NumberORId>}
<print> ::= print_line ("<string>");
<number> ::= <digit> {<digit>}
<String> ::= <letter> {<letter>}
<conditional op> ::= < | > | == | != | <= | >=
<Arithematic OP> ::= + | - | * | /
```

Output



```
PS D:\Handong\4-1\Compiler Theory\HW2> javac RecurParser.java
PS D:\Handong\4-1\Compiler Theory\HW2> java RecurParser .\testInputsForRecurParser\test1.txt
Parsing OK
PS D:\Handong\4-1\Compiler Theory\HW2> java RecurParser .\testInputsForRecurParser\test1error.txt
Parsing Failed
end not found
PS D:\Handong\4-1\Compiler Theory\HW2> java RecurParser .\testInputsForRecurParser\test2.txt
Parsing OK
PS D:\Handong\4-1\Compiler Theory\HW2> java RecurParser .\testInputsForRecurParser\test2error.txt
Parsing Failed
end not found
PS D:\Handong\4-1\Compiler Theory\HW2> java RecurParser .\testInputsForRecurParser\test3.txt
Parsing OK
PS D:\Handong\4-1\Compiler Theory\HW2> java RecurParser .\testInputsForRecurParser\test3error.txt
Parsing Failed
begin not found
```