# Team 3 Tiny-Ada Syntax Analyzer 설명서

What is the purpose and function of this analyzer?

* Python으로 Ada Syntax analyzer의 일부를 구현하였다. text input이 Ada 문법(의 일부), 즉 Tiny Ada 문법에 부합하는지 확인한다.
* charIo로 text input을 받아온 후, Scanner를 통해 해당 input을 Token 별로 나눈다. Parser를 통해 해당 Token들로 구성된 text input이 TinyAda의 문법에 부합하는지 확인한다. (Parse)

How to Compile? How to Use?

* 터미널에서, python TerminalApp.py로 실행

What does it consist of?

* Token class, CharIO class, Scanner class, Parser class

What is the role of each class?

* Please read below.

Following 4 design points (from textbook)

1. Merge the processing for both rules into a single method
2. Scanner method nextToken must be run once to obtain the first token
3. Each parsing method leaves behind the next token following the phrase just processed
4. Each parsing method expects no parameters and returns no value.

## Token class

* AND, ARRAY, DIV, PLUS, IF, WHILE 등과 같은 기본적인 operator 및 Symbol을 사전 정의하였다.

Token(), Token(newcode) : 생성자. AND, ARRY, DIV 처럼 해당 operators/Symbols의 code를 가진 Token을 생성한다.

## CharIO class

* Text file input을 받아, 해당 text를 소문자로 저장한 후, 각 줄 혹은 글자를 반환한다.

charIO(stream): Constructor. file name을 parameter로 넣으면, 해당 file의 text를 읽어 들인다.

reset(self): text file의 처음 위치로 돌아간다.

putError(message): 실행 도중 error가 일어날 경우 해당 error message를 출력한다.

reportErrors(): Compilation 이후 총 에러 개수를 출력한다.

getChar(): 현재 위치의 char를 반환한다

nextline(): 다음 줄을 반환한다

readFile(stream): charIO가 생성될 때 들어온 parameter, 즉 file name을 open하고 소문자로 읽어들여 클래스 변수 sourceprogram에 저장한다.

getLine(): 현재 Line을 반환한다.

Print(), makespaces(number): error message를 출력할 때 사용된다.

## Scanner class

* CharIO로 받아온 text input을 Token 별로 쪼갠다.

Scanner(c): Constructor. token별로 쪼개고자 하는 charIO(즉, text input)를 parameter로 전달한다.

reset(): text input에서 처음 위치로 돌아간다.

initKeywords(): 사전 정의된 Keyword Token(AND, ARRAY 등)을 리스트에 저장한다.

initSingleOps(): 사전 정의된 Operator Token(:, = 등)을 리스트에 저장한다.

initDoubleOps(): 사전 정의된 Double operator Token(\*\*, >= 등)을 리스트에 저장한다.

findToken(table, target): table에서 target token을 찾아 반환한다. 찾지 못할 경우 error token을 반환한다.

skipBlanks(): 읽어온 text input이 blank일경우 skip하고 다음 input으로 넘어간다.

getIdentifierOrKeyword(): identifier나 keyword 규칙을 통해 text input 중 identifier나 keyword를 찾는다.

getInteger(): # 이후에 오는 letter나 digit을 찾는다.

getBasedInteger(): Based integer를 찾는다.

charToInt(): char를 int로 바꾼다.

getCharacter(): '''를 찾는다.

getDoubleOp(), getSingleOP(): single 혹은 double operator를 얻어온다.

nextToken(): 다음 token으로 넘어간다.

## Parser class

* scanner로 얻어온 token들이 문법에 걸맞는지 확인하고, 오류가 있을 경우 검출한다.

Parser(c,s): constructor. c는 charIO object, s는 Scanner object

reset(): scanner를 reset한다

initHandles(): add, multiply, relational, declaration, statement operator 종류의 token들을 해당하는 set에다 넣는다.

accept(expected, errorMessage): 해당 token이 기대했던 것이 아닌 경우 error를 일으킨다. 기대한 것일 경우, 다음 token으로 넘어간다.

fatalError(errorMessage): accept에서, error을 일으키기 위해 사용되는 함수다.

parse(): 전체 코드 진행

* subprogramBody(), subprogramSpecification(), formalPart(), ParameterSpecification(), mode(), declarativePart(), basicDeclaration(), numberofObjectDeclaration() : Text Input이 Ada에서 요구하는 Part 순서로 이루어져있는지 확인한다

## \*참고. Tiny Ada의 EBNF 문법

= means "is def ined as"

" " enclose l i teral i tems

[ ] enclose i tems which may be omi t ted

{ } enclose i tems which may appear zero or more t imes

| indicates a choice

( ) are used for grouping requi red choices

< > enclose semant ic qual i f icat ions

subprogramBody =

subprogramSpeci f icat ion "is"

declarat ivePar t

"begin" sequenceOfStatements

"end" [ <procedure>ident i f ier ] ";"

declarat ivePar t = { basicDeclarat ion }

basicDeclarat ion = objectDeclarat ion | numberDeclarat ion

| typeDeclarat ion | subprogramBody

objectDeclarat ion =

ident i f ierList ":" typeDef ini t ion ";"

numberDeclarat ion =

ident i f ierList ":" "constant" ":=" <stat ic>expression ";"

ident i f ierList = ident i f ier { "," ident i f ier }

typeDeclarat ion = "type" ident i f ier "is" typeDef ini t ion ";"

typeDef ini t ion = enumerat ionTypeDef ini t ion | arrayTypeDef ini t ion

| range | <type>name

range = "range " simpleExpression ". ." simpleExpression

index = range | <type>name

enumerat ionTypeDef ini t ion = "(" ident i f ierList ")"

arrayTypeDef ini t ion = "array" "(" index { "," index } ")" "of" <type>name

subprogramSpeci f icat ion = "procedure" ident i f ier [ formalPar t ]

formalPar t = "(" parameterSpeci f icat ion { ";" parameterSpeci f icat ion } ")"

parameterSpeci f icat ion = ident i f ierList ":" mode <type>name

mode = [ "in" ] | "in" "out" | "out"

sequenceOfStatements = statement { statement }

statement = simpleStatement | compoundStatement

simpleStatement = nul lStatement | assignmentStatement

| procedureCal lStatement | exi tStatement

compoundStatement = i fStatement | loopStatement

nul lStatement = "nul l" ";"

assignmentStatement = <var iable>name ":=" expression ";"

i fStatement =

"i f" condi t ion "then" sequenceOfStatements

{ "elsi f" condi t ion "then" sequenceOfStatements }

[ "else" sequenceOfStatements ]

"end" "i f" ";"

loopStatement =

[ i terat ionScheme ] "loop" sequenceOfStatements "end" "loop" ";"

i terat ionScheme = "whi le" condi t ion

exi tStatement = "exi t" [ "when" condi t ion ] ";"

procedureCal lStatement = <procedure>name [ actualParameterPar t ] ";"

actualParameterPar t = "(" expression { "," expression } ")"

condi t ion = <boolean>expression

expression = relat ion { "and" relat ion } | { "or" relat ion }

relat ion = simpleExpression [ relat ionalOperator simpleExpression ]

simpleExpression =

[ unaryAddingOperator ] term { binaryAddingOperator term }

term = factor { mul t iplyingOperator factor }

factor = pr imary [ "\*\*" pr imary ] | "not" pr imary

pr imary = numer icLi teral | st r ingLi teral | name | "(" expression ")"

name = ident i f ier [ indexedComponent ]

indexedComponent = "(" expression { "," expression } ")"

relat ionalOperator = "=" | "/=" | "<" | "<=" | ">" | ">="

binaryAddingOperator = "+" | "-"

unaryAddingOperator = "+" | "-"

mul t iplyingOperator = "\*" | "/" | "mod"