Define our own language!

compiler

Homework #10

TeamName: ThinkTank

student number:

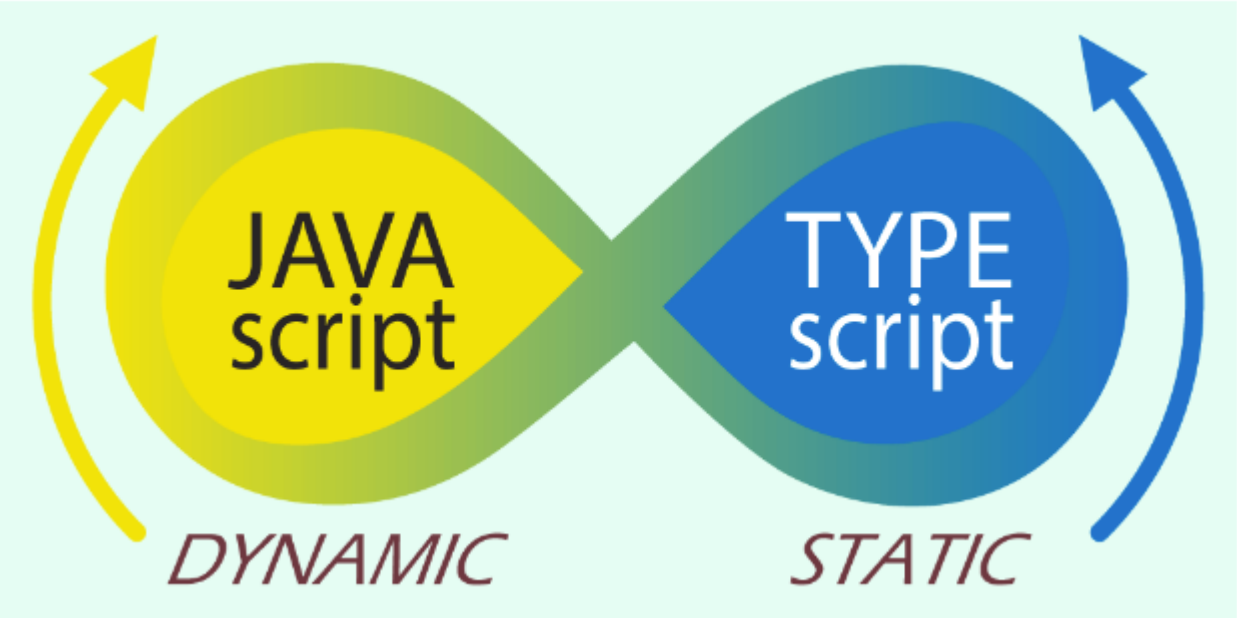
201228174: 안대희 **201424492:** 여종현

1. introduction

1.1 A base language

The base language is **Python**. Python is a readable and accessible language. The development of artificial intelligence libraries such as TensorFlow and keras are constantly expanding Python's share. However, since it is a weak type language, there is a drawback that the python program does not correctly catch hazard errors on run time. In fact, while we were developing a robot simulator in Python, we spent a lot of time catching errors at runtime. that would not have happened if we had strongly typed language. On the other hand, the strong type languages correctly detects type errors much before code execution, forces developers to clearly specify types. So, to implement Python as a strong type language, We suggest **'TypeSun'** (python-type combination)

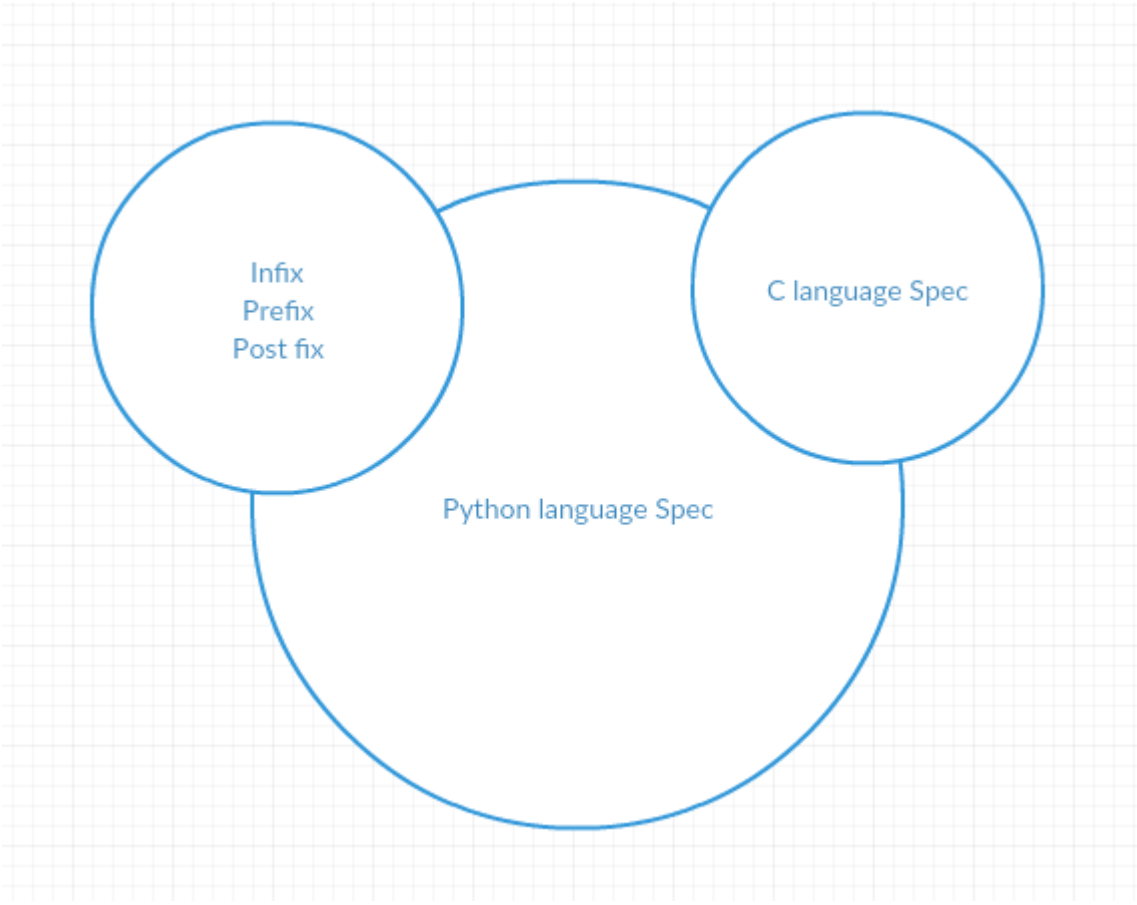
1.1.1 background knowledge



Before we explain the project we are going to do, I will first describe the language called JavaScript. JavaScript and Python are both dynamic languages, and they contain many similar features. And JavaScript has the same problem with the static language that Python has. There is a language that solve the problem of JavaScript elegantly, called **Typescript**, which was a strongly typed language and greatly improved the

problems of the dynamic nature of JavaScript. Typescripts, add type stability to the productivity of JavaScript, is currently being used for implementing STL porting on several open sources.

1.2 Improved feature of TypeSun



(The smaller the number, the higher the priority.)

1. A Strong type langauge (It can specify type!)
2. There are also return types in function definition.
3. Logical operators available (Existing python is available only for and and or, but typeSun allows &&, ||, ! and some bit operator)
4. Not only infix expressions, but also prefix expressions and postfix expressions can be compiled. [1] This is a toy grammar that makes it easier to port and execute things written in other languages such as lisp into Python.
5. Add a Output function called 'out'. This preserves the intuitive appearance of the output without the need for a newline symbol. [2]

2. Syntex EBNF

Key Issues: Prior to making EBNF, we looked at two specs of language that we decided to converge. The specs for c langauge and python 3.71 were easily found on the internet, and that spec was simpler than we thought.

1. Maintains basic Python syntax. But we have Notation specifier.

single_input: NEWLINE | notation_specifier simple_stmt
| simple_stmt
| notation_specifier compound_stmt NEWLINE | compound_stmt NEWLINE

file_input: (NEWLINE | notation_specifier stmt | stmt)* ENDMARKER

eval_input: testlist NEWLINE* ENDMARKER

suite: (notation_specifier simple_stmt | simple_stmt)| NEWLINE INDENT (notation_specifier stmt | stmt)+ DEDENT (This syntex is one of the most basic and essential grammar of python)

notation_specifier: 'pre:' | 'in:' | 'post:'

1. Add type_specifier with adjusting typeSun.

Numeric Types — int, float, complex

Sequence Types — list, tuple, range

Text Sequence Type - str

Binary Sequence Types - bytes, bytearray, memoryview

Set Types - set, frozenset

Mapping Types

classdef: 'class' NAME ['(' [arglist] ')'] ':' suite

arglist: argument (',' argument)* [',']

argument: (test [comp_for] | test '=' test | '**' test | '*' test)

funcdef: 'def' type_specifier NAME parameters ['->' test] ':' suite

parameters: '(' [typedargslist] ')'

typedargslist: (tfpdefVal ['=' test] (',' tfpdefVal ['=' test])* [',' ['' *[tfpdefVal] (',' tfpdefVal ['=' test])* [',' [**'' tfpdefVal [',']]] | ''** tfpdefVal [',']]] | '' *[tfpdefVal] (',' tfpdefVal ['=' test])* [',' [**'' tfpdefVal [',']]] | ''** tfpdefVal [','])

Point!: TypeSun explicitly check the type by adding tfpedfVal in the existing Python syntax

tfpdefVal: type_specifier NAME [':' test]

type_specifier : INT | FLOAT
| COMPLEX_SPECIFIER
| LIST
| TUPLE
| RANGE
| BYTES
| BYTEARRAY
| MEMORYVIEW
| SET
| DICT

TEST_COMPLEX_SPECIFIER : (INT | FLOAT)+ (INT | FLOAT)+ 'j' ;

EXPRLIST: (expr|star_expr) (',' (expr|star_expr))* [',']

TESTLIST: test (',' test)* [',']

EXPRTUPLE: (expr|star_expr) (',' (expr|star_expr))* [',']

TESTTUPLE: test (',' test)* [',']

And so on.

3. Add C like type_specifier with adjusting typeSun.

or_test: and_test (('or' | '||') and_test)*

and_test: not_test (('and' | '&&') not_test)*

not_test: ('not'| '!') not_test | comparison

comparison: expr (comp_op expr)*

comp_op: '<'|'>'|'=='|'>='|'<='|'<>'|'!='|'in'|'not' 'in'|'is'|'is' 'not'

star_expr: '*' expr

expr: xor_expr ('|' xor_expr)*

xor_expr: and_expr ('^' and_expr)*

and_expr: shift_expr ('&' shift_expr)*

shift_expr: arith_expr (('<<'|'>>') arith_expr)*

arith_expr: term (('+'|'-') term)*

term: factor (('*'|'@'|'/'|'%'|'//') *factor)*

factor: ('+'|'-'|'~') factor | power

power: atom_expr ['**' factor]

atom_expr: ['await'] atom trailer*

atom: ('(' [yield_expr|testlist_comp] ')' | '[' [testlist_comp] ']' | '{' [dictorsetmaker] '}' | NAME | NUMBER | STRING+ | '...' | 'None' | 'True' | 'False')

testlist_comp: (test|star_expr) (comp_for | (',' (test|star_expr))* [','])

trailer: '(' [arglist] ')' | '[' subscriptlist ']' | '.' NAME

subscriptlist: subscript (',' subscript)* [',']

subscript: test | [test] ':' [test] [sliceop]

sliceop: ':' [test]

3. Code Example

```
def int mul(int a, int b)
    return a*b;

def main()
    int a=0, b = 20, sum = 0, mul = 0;
    int
        a = %a
        b = %b

    post: sum = a b +
    mul = mul(a,b) # default behavior

    if(sum>=0 && a>=0 && b>=0):
        out:
            a와 b 모두 양수이다.
            a=%a, b=%d이고
            a + b = %c, a * b = %mul 이다.
    else:
        out:
            a와 b 모두 양수가 아닐수도 있다.
```

```
input

a = 10
b = 20
```

```
output

a 와 b 모두 양수이다.
a = 10, b = 20 이고
a + b = 30, a * b = 200 이다
b = 20
```

4. Reference sites

http://www.appmonks.net/id/93/Typescript-VS-JavaScript/ http://www.wdz.eng.br/Python3DsBnf.htm
https://docs.python.org/3/reference/grammar.html
https://cs.wmich.edu/~gupta/teaching/cs4850/sumII06/The%20syntax%20of%20C%20in%20Backus-Naur%20form.htm

5. Other

It will be maintained on JongHyeonYeo's github https://github.com/mindgitrwx/compiler_prototype

[1] In the cases of prefix, infix and postfix notation, there are time and space limits to consider all EBNFs for each, so in this prototype document, we specified a document by extracting the parts related to type.

[2] As far as IO handling, We cannot distinguish between whether it is directly related of making compiler or of making library function, so We have created a prototype of EBNF, focusing on the above four points.