# Making a Domain Specific Language

***Specific Language for Music Player***

심우진 (Shim Woo jin)
송종현 (Song Jong hyun)
최준영 (Choi Jun young)

**How to Create a Programming Language**
**Professor Antonino N.Mione**

# Contents

**Theorem**

What is DSL(Domain Specific Language)?

What is Parsing?

How to make a Parser

# Contents

**Project**

What is our Domain?

Making a Syntax

Making a Parser

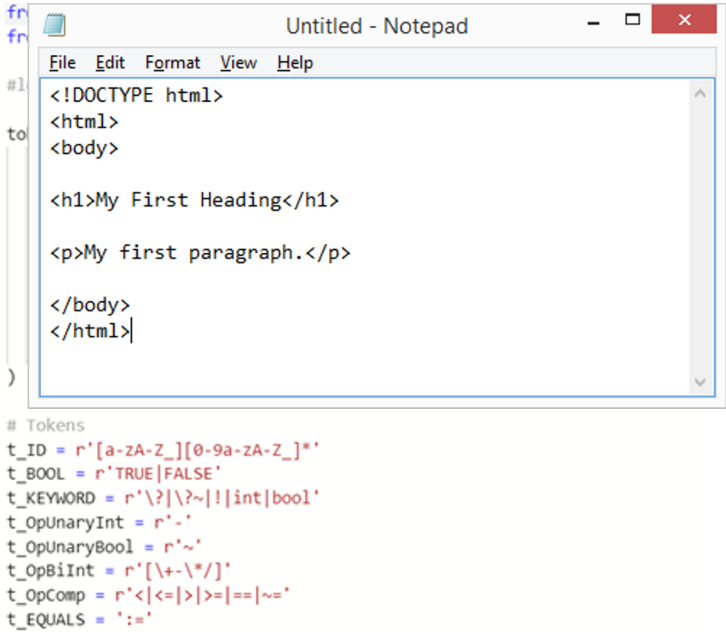
Result

# What is DSL?

Domain Specific Language is a language that specialized in a particular domain(ex. Industry, Web pages…)

HTML is an example of DSL, it is used for making or using Web Site.

It has the advantage of being easy to use in certain areas.




To make a language, we have to define the ‘keyword’ and ‘grammer’.

Next, use scanner to scan languages to ‘Token’

And then we make a parser to make parsing program.

# What is parsing?

First, token is a basic language no longer grammatically divisible. (ex. operator, integer, digit…)

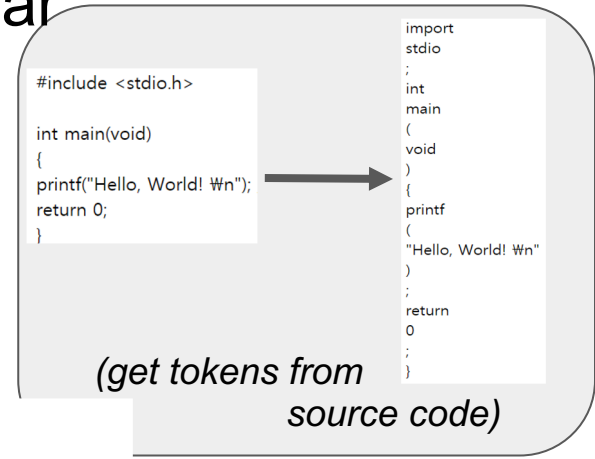
Parsing is the process of analyzing a string to grammar.

In programming language, parsing is a process of complie or interpret that combine tokens to get datas what we want and check the grammar of language.

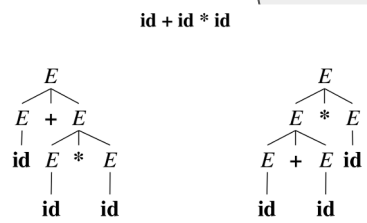
# What is parsing?

How parser get datas and check the grammar

1) Get tokens from scanning process




1) Make Syntax tree by tokens




1) Check the grammar by Syntax tree

# What is our Domain

Our goal

Making specific language for Music Player


Functions that going to make

- Add or remove music to player (Title ： number and character)
- Sort music in player randomly
- Add music randomly
- Let people know how they listen music (Time ： [num]:[num]:[num])
- Make random playlist

# Making a Syntax

select statement(like ‘if’ in high-level-language)

- Why : there are many commands that users can do.

iteration statement(like ‘while’ in high-level-language)

- Why : Wait for a response to answer your question.

function call

- Why : We should resolve various orders from users.

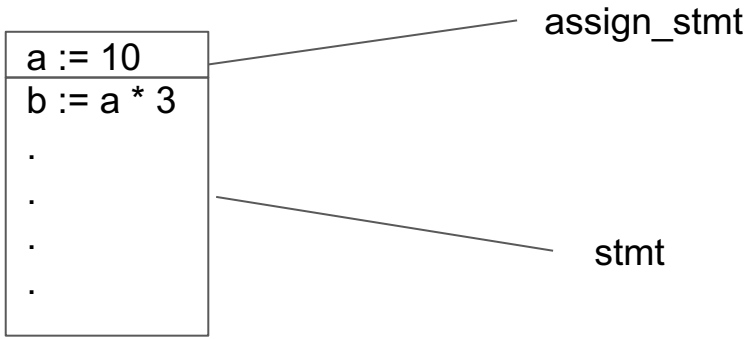
Operator : comp, bool, +,-,*,/, ..

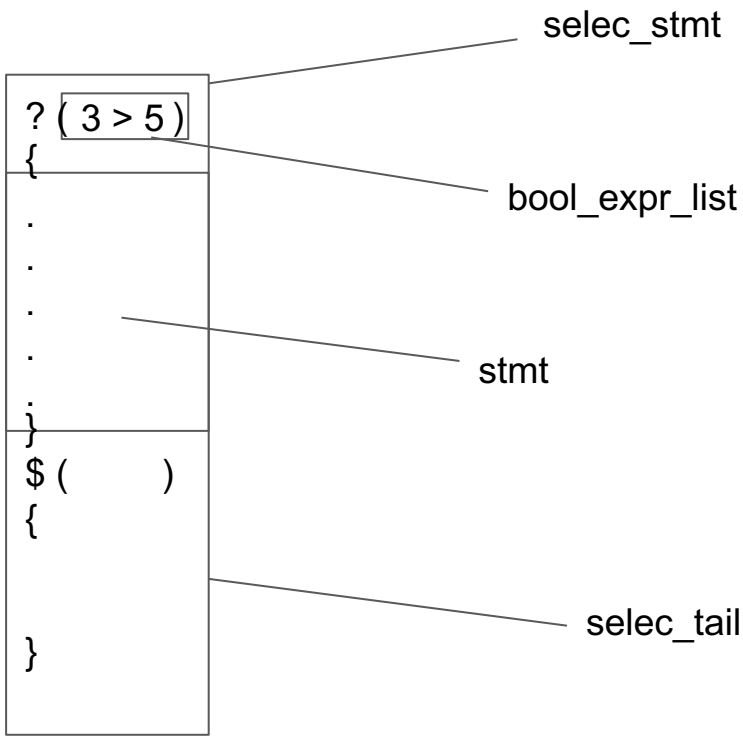
# Making a Syntax

```
start          =>  stmt
stmt           =>  selec_stmt stmt
                 | iter_stmt stmt
                 | assign_stmt stmt
                 | func_call_stmt stmt
                 | e
selec_stmt     =>  ? ( bool_expr_list ) { stmt } selec_tail
selec_tail     =>  $ { stmt }
                 | e
iter_stmt      =>  ! ( bool_expr_list ) { stmt }
assign_stmt    =>  ID := int_expr_list
func_call_stmt =>  FUNC ( comma_expr_list )
                 | FUNC ( )
comma_expr_list=>  comma_expr_list , int_expr_list
                 | comma_expr_list , bool_expr_list
                 | int_expr_list
                 | bool_expr_list
```

```
bool_expr_list =>  bool_expr_list binary_bool_opt bool_expr
                 | bool_expr
                 | unary_bool_opt ( bool_expr_list )
                 | bool_expr_list binary_bool_opt expr comp_opt expr
bool_expr      =>  ID
                 | BOOL
                 | int_expr comp_opt int_expr
int_expr_list  =>  int_expr_list '+' int_term_list
                 | int_expr_list '-' int_term_list
                 | int_term_list
int_term_list  =>  int_term_list '*' int_fact
                 | int_term_list '/' int_fact
                 | int_fact_list
int_fact       =>  ( int_expr_list )
                 | ID
                 | INTEGER
```

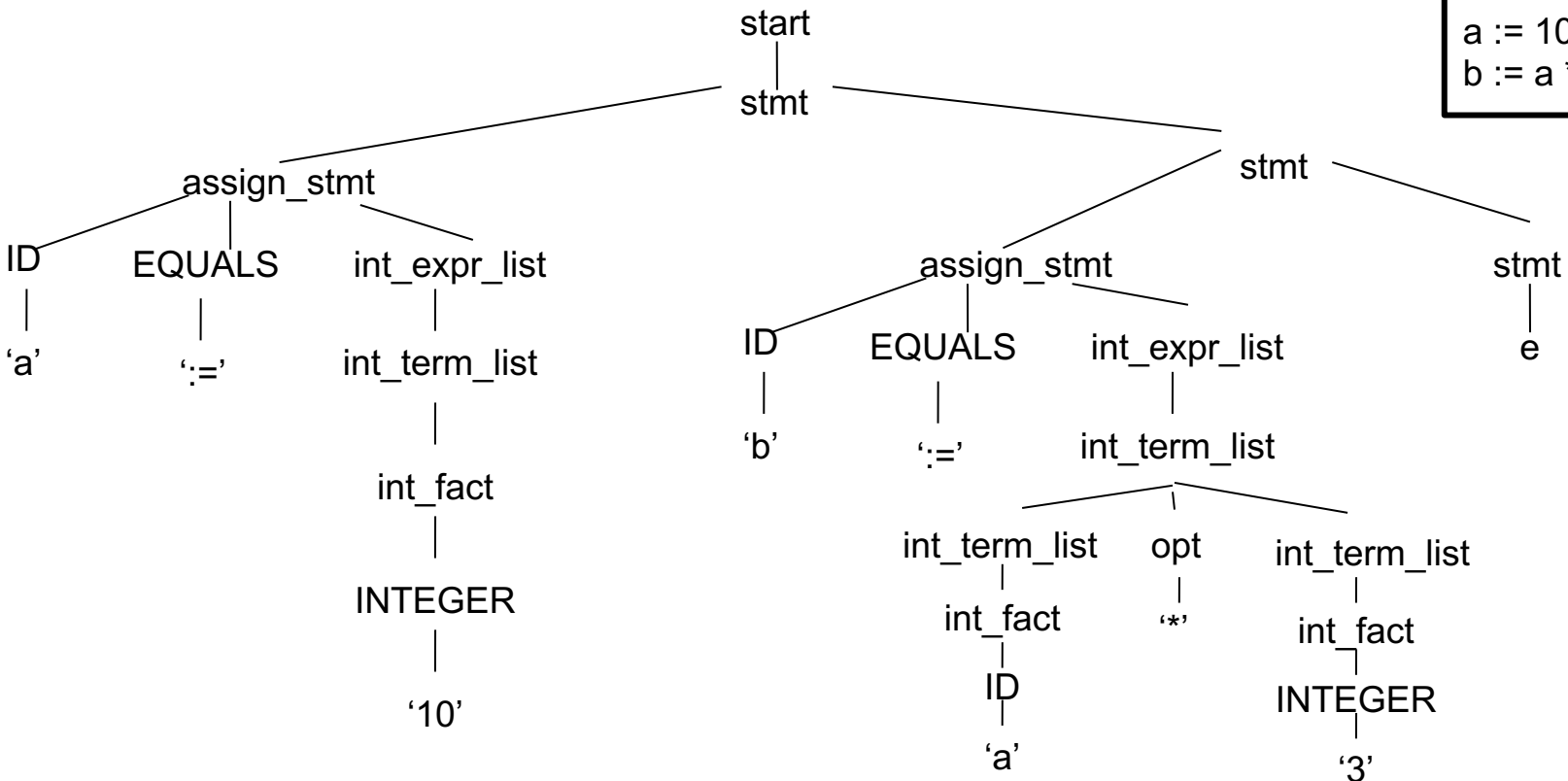
# Our Syntax


a := 10
b := a * 3
.
.
.
.
assign_stmt
stmt


? ( 3 > 5 )
{
.
.
.
.
.
}
$ ( )
{
}
selec_stmt
bool_expr_list
stmt
selec_tail

# Syntax tree for our language

```
a := 10
b := a * 3
```



# Making a parser - lexer, token

```
from DSL.ply_master.ply import lex
from DSL.ply_master.ply import yacc

# lexer

tokens = (
    'ID', 'BOOL', 'INTEGER', 'NAME',
    'IF', 'ELSE', 'WHILE',
    'OpUnaryBool',
    'ADD', 'MUL',
    'OpBiBool',
    'OpComp',
    'LPAREN', 'RPAREN',
    'LBRACKET', 'RBRACKET',
    'EQUALS',
    'KEYWORD',
    'FUNC',
    'COMMA'
)
```

```
# Tokens
t_FUNC = r'InputNum|InputString|ReadFile|WriteFile|PrintList|PrintString|UpdateSong|ModifySong|AppendField|NewList'
t_NAME = r'\"([a-zA-Z0-9_ \(\)]+\.[a-z0-9]+|[a-zA-Z0-9_ \(\)]+)\"'
t_KEYWORD = r'INT|BOOL'
t_IF = r'\?'
t_ELSE = r'\$'
t_WHILE = r'!'
t_OpUnaryBool = r'~'
t_ADD = r'[\+\-]'
t_MUL = r'[\*/]'
t_OpComp = r'<|<=|>|>=|==|~='
t_EQUALS = ':='
t_OpBiBool = r'[|&\^]'
t_COMMA = r','

t_ignore = ' \t'
```

```
def t_ID(t):
    r'[a-z_][a-zA-Z0-9_]*'
    print("'ID', " + t.value)
    return t

def t_INTEGER(t):
    r'[0-9]+'
    t.value = int(t.value)
    print("'INTEGER', " + str(t.value))
    return t

def t_BOOL(t):
    r'TRUE|FALSE'
    if t.value == 'TRUE': t.value = True
    else: t.value = False
    print("'BOOL', " + str(t.value))
    return t

def t_LPAREN(t):
    r'\('
    print("'LPAREN', '('")
    return t

def t_RPAREN(t):
    r'\)'
    print("'RPAREN', ')'")
    return t

```

# Making a parser - parser code

```
93  def p_start(p):
94      'start : stmt'
95      print('start')
96      p[0] = p[1]
97
98
99  def p_stmt(p):
100     '''stmt : selec_stmt stmt
101             | iter_stmt stmt
102             | assign_stmt stmt
103             | func_call_stmt stmt
104             | '''
105     print("stmt")
106     #if len(p) == 3: p[0] = p[1] + p[2]
107
108
109 def p_selec_stmt(p):
110     'selec_stmt : IF LPAREN bool_expr_list RPAREN LBRACKET stmt RBRACKET selec_tail'
111     print('selec_stmt')
112     if p[1] == '?':
113         if p[3]:
114             p[0] = p[6]
115         else:
116             p[0] = p[8]
117     else:
118         print('Error Token')
119
120
```

```
128 def p_iter_stmt(p):
129     'iter_stmt : WHILE LPAREN bool_expr RPAREN LBRACKET stmt RBRACKET'
130     print('iter_stmt')
131     if p[1] == '!':
132         if p[3] == True:
133             p[0] = p[6]
134     else:
135         print('Error Token')
136
137
138 def p_assign_stmt(p):
139     '''assign_stmt : ID EQUALS int_expr_list'''
140     print('assign_stmt')
141     if not (p[1] in names): names.update({p[1]:0})
142     names[p[1]] = p[3]
143
144 def p_func_call_stmt(p):
145     '''func_call_stmt : FUNC LPAREN comma_expr_list RPAREN
146                       | FUNC LPAREN RPAREN'''
147     print('func_call_stmt')
148
149
150 def p_comma_expr_list(p):
151     '''comma_expr_list : String
152                        | String COMMA String COMMA String
153                        | String COMMA String COMMA String COMMA String'''
154
155 def p_String(p):
156     '''String : NAME
157               | func_call_stmt'''
158
```

# Making a parser - parser code

```
159  def p_bool_expr_list(p):
160      '''bool_expr_list : bool_expr_list OpBiBool bool_expr
161                        | bool_expr
162                        | OpUnaryBool LPAREN bool_expr_list RPAREN
163                        | KEYWORD LPAREN int_expr_list RPAREN'''
164      print('bool_expr_list')
165      if len(p) == 2:
166          p[0] = p[1]
167      elif len(p) == 4:
168          if p[2] == '|':
169              p[0] = p[1] or p[3]
170          elif p[2] == '&':
171              p[0] = p[1] and p[3]
172          elif p[2] == '^':
173              p[0] = p[1] ^ p[3]
174      elif len(p) == 5:
175          if p[2] == 'bool':
176              p[0] = bool(p[4])
177          else:
178              p[0] = not p[3]
```

```
200  def p_bool_expr(p):
201      '''bool_expr : int_expr_list OpComp int_expr_list
202                   | BOOL'''
203      print('bool_expr')
204      if type(p[1]) == str:
205          if not p[1] in names: names.update({p[1]:0})
206          p[0] = names[p[1]]
207      else:
208          p[0] = p[1]
209
210
211  def p_int_expr_list(p):
212      '''int_expr_list : int_expr_list ADD int_term_list
213                       | int_term_list'''
214      print('int_expr_list')
215      if len(p) == 2:
216          p[0] = p[1]
217      elif p[2] == '+':
218          p[0] = p[1] + p[3]
219      elif p[2] == '-':
220          p[0] = p[1] - p[3]
221
222
223  def p_int_term_list(p):
224      '''int_term_list : int_term_list MUL int_fact
225                       | int_fact'''
226      print('int_term_list')
227      if len(p) == 2:
228          p[0] = p[1]
229      elif p[2] == '*':
230          p[0] = p[1] * p[3]
231      elif p[2] == '/':
232          p[0] = p[1] / p[3]
```

# Result

Let’s show the parsing process!