CSC 326 GROUP 16.

P15/1637/2019: ABEL BENARD MWENDWA.
P15/1639/2019: MURITHI CHRIS.
P15/81770/2017: BOSIRE SANDRAH KWAMBOKA
P15/137631/2019: ALI AMINA ABDI
P15/37269/2016: JAMA WARSAME MOHAMUD

1.1 Grammar description

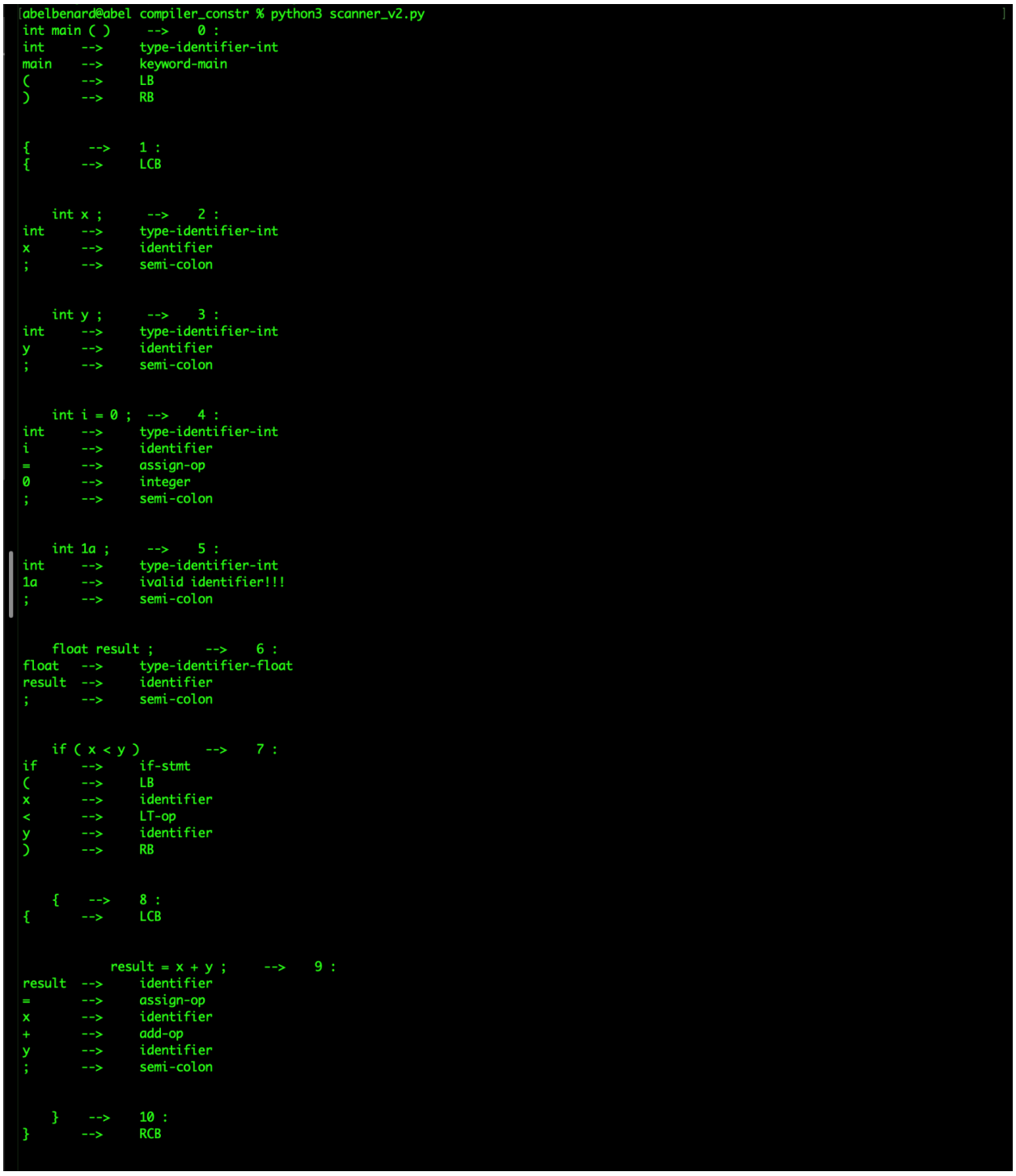
- <program> ::= <int main () {declarations statements} >
- <declarations > ::= <declaration>
- <declaration> ::= type identifier
- <type> ::= <integer> | <float>
- <integer>::=<digit> | <digit>
- <float> ::= <integer>.<integer>
- <digit>:: = **0 | … | 9**
- <Identifier>:: = **a | … | z A | … | Z 0 | … | 9**
- <statements> ::= <statement>
- <statement> ::= <block> | <control-stmt> | <expression> | <ret-stmt>
- <block> := <statements>
- <control-stmt> ::= <if> |< while>
- <if> ::= **if** (<expression>) <statement>
- <while> ::= **while** (<expression>) <statement >
- <expression> ::= <arithmetic> | <relational> | <conditional> | <logical> | <assignment>
- <arithmetic> ::= <add-expr> | <mult-expr>
- <add-expr>::= <integer> <add-op> <integer>| <float><add-op><float>
- <add-op> ::**= + | -**
- <mult-expr> ::= <integer><mult-op><integer> | <float><mult-op><float>
- <mult-op> ::= *** | /**
- <relational> ::= <equ-expr> | <rel-expr>
- <equ-expr> ::= <integer><equ-op><integer> |
  <float><equ-op><float>|<identifier><equ-op><identifier>
- <equ-op> ::= **== | !=**
- <rel-expr> ::= <integer> <rel-op> <integer> | <integer><rel-op> <integer>
- <rel-op> ::= **< | >| <= |>=**
- <logical> ::= <relational> <logic-op> <relational>
- <logic-op> ::= **&& | ||**
- <assignment> ::= <identifier> = <expression>
- <ret-stmt> ::= **return** <integer>

1.2 Sample Code

```
int main ( )
{
    int x ;
    int y ;
    float result ;

    if ( x < y )
    {
        result = x + y ;
    }
    if ( x > y )
    {
        result = x - y ;
    }
    return 0 ;
}
```

1.3 scanner output for sample code (including screenshots of the scanner output)



```
[abelbenard@abel compiler_constr % python3 scanner_v2.py                        ]
int main ( )     -->    0 :
int     -->     type-identifier-int
main    -->     keyword-main
(       -->     LB
)       -->     RB


{        -->    1 :
{       -->     LCB


    int x ;      -->    2 :
int     -->     type-identifier-int
x       -->     identifier
;       -->     semi-colon


    int y ;      -->    3 :
int     -->     type-identifier-int
y       -->     identifier
;       -->     semi-colon


    int i = 0 ;  -->    4 :
int     -->     type-identifier-int
i       -->     identifier
=       -->     assign-op
0       -->     integer
;       -->     semi-colon


    int 1a ;     -->    5 :
int     -->     type-identifier-int
1a      -->     ivalid identifier!!!
;       -->     semi-colon


    float result ;       -->    6 :
float   -->     type-identifier-float
result  -->     identifier
;       -->     semi-colon


    if ( x < y )         -->    7 :
if      -->     if-stmt
(       -->     LB
x       -->     identifier
<       -->     LT-op
y       -->     identifier
)       -->     RB


    {    -->    8 :
{       -->     LCB


            result = x + y ;     -->    9 :
result  -->     identifier
=       -->     assign-op
x       -->     identifier
+       -->     add-op
y       -->     identifier
;       -->     semi-colon


    }    -->    10 :
}       -->     RCB
```

```
    if ( x > y )        -->    11 :
if      -->     if-stmt
(       -->     LB
x       -->     identifier
>       -->     GT-op
y       -->     identifier
)       -->     RB


    {    -->    12 :
{       -->     LCB


            result = x - y ;     -->    13 :
result  -->     identifier
=       -->     assign-op
x       -->     identifier
-       -->     sub-op
y       -->     identifier
;       -->     semi-colon


    }    -->    14 :
}       -->     RCB


         -->    15 :


    return 0 ;   -->    16 :
return  -->     return-stmt
0       -->     integer
;       -->     semi-colon


}        -->    17 :
}       -->     RCB
```

1.4 Description of parsing strategy chosen and justification

Predictive LL1 top down parsing - where we begin at the start symbol and try to predict the next production to apply by the use of lookahead token were we look at some number of tokens of the input to help make the decision of the next production to use until we end up at the users program .
Justification
Predictive parser is fast and easier to implement

1.5 Parser code

```
import re


grammar = {
   "type-identifier-int" : "int",
   "type-identifier-float" : "float",
   "keyword-main" : "main",
   "if-stmt" : "if",
```

```
    "while-stmt" : "while",
    "return-stmt" : "return",
    "add-op" : "+",
    "sub-op" : "-",
    "mul-op" : "*",
    "div-op" : "/",
    "GT-op" : ">",
    "LT-op" : "<",
    "assign-op" : "=",
    "equ-op" : "==",
    "not_equ-op" : "!=",
    "logical-and" : "&&",
    "logical-or" : "||",
    "LCB" : "{",
    "RCB" : "}",
    "LB" : "(",
    "RB" : ")",
    "semi-colon" : ";"
}


code_file = "sample-code.txt"
# code_file = "sampleerrorcode.txt"

with open(code_file, "r") as fo:
    sample_code = fo.read()

phrase = sample_code.replace("\n", " ")
phrase = phrase.split()


def prog(tok, next_tok):
    if next_tok == "main":
        if phrase[(phrase.index(next_tok) + 1)] == "(":
            if phrase[(phrase.index(next_tok) + 2)] == ")":
                if phrase[(phrase.index(next_tok) + 3)] == "{":
                    if phrase[-1] == "}":
                        print("<Program>")
                    else:
                        print("'}' expected at the end of the program")
                else:
                    print("'{' expected after 'int main ()'")
```

```
def declarations(tok, next_tok):
   floating_point = re.search(r'[0-9]([0-9])*.[0-9]([0-9])*', f'{next_tok}')
   integer =  re.search(r'[0-9]([0-9])*', f'{next_tok}')
   identifiers = re.search(r'[A-Za-z]([A-Za-z]|[0-9])*', f'{next_tok}')
   inv_id = re.search(r'[0-9][A-Za-z]([A-Za-z]|[0-9])*', f'{next_tok}')
   inv_id2 = re.search(r'[A-Za-z]([A-Za-z])*.[0-9]([0-9])*', f'{next_tok}')
   initialisation = re.search(r'[A-Za-z]([A-Za-z]|[0-9])*=[0-9]([0-9])*;',
f'{next_tok}{phrase[(phrase.index(next_tok) + 1)]}{phrase[(phrase.index(next_tok) +
2)]}{phrase[(phrase.index(next_tok) + 3)]}')

   string = f"{tok} {next_tok} {phrase[(phrase.index(next_tok) + 1)]}"
   if inv_id:
       print(f"\t<Declaration> : {string}")
       print(f"\t\t{inv_id[0]}\t-->\tidentifier expected : identifiers can not start
with a number")
       return
   if inv_id2:
       print(f"\t<Declaration> : {string}")
       print(f"\t\t{inv_id2[0]}\t-->\tinvalid identifier!!!")
       return
   if integer:
       print(f"\t<Declaration> : {string}")
       print(f"\t\t{integer[0]}\t-->\tidentifier expected")
       return
   if floating_point:
       print(f"\t<Declaration> : {string}")
       print(f"\t\t{floating_point[0]}\t-->\tidentifier expected")
       return
   if initialisation:
       string += phrase[(phrase.index(next_tok) + 2)]
       string += phrase[(phrase.index(next_tok) + 3)]
       print(f"\t<Declaration> : {string}")
   if identifiers:
       if phrase[(phrase.index(next_tok) + 1)] == ";":
           print(f"\t<Declaration> : {string}")
           return


def assignment(tok, tok_index):
   identifiers = re.search(r'[A-Za-z]([A-Za-z]|[0-9])*', f'{phrase[(tok_index + 4)]}')
```

```
   keyword = re.search(r'int|float|if|while|return|main', f'{tok}')
   assign_expr0 = re.search(r'[A-Za-z]([A-Za-z]|[0-9])*=[0-9]([0-9])*',
f'{tok}{phrase[(tok_index + 1)]}{phrase[(tok_index + 2)]}')
   assign_expr1 = re.search(r'[A-Za-z]([A-Za-z]|[0-9])*=[A-Za-z]([A-Za-z]|[0-9])*',
f'{tok}{phrase[(tok_index + 1)]}{phrase[(tok_index + 2)]}')


   if keyword:
       return
   if assign_expr0:
       # print(f"valid assign {tok}")
       if phrase[(tok_index + 3)] == ";":
           print(f"\t\t<Statement> : {tok} {phrase[(tok_index + 1)]}
{phrase[(tok_index + 2)]} {phrase[(tok_index + 3)]}")
       elif phrase[(tok_index + 3)] == "+" or phrase[(tok_index + 3)] == "-" or
phrase[(tok_index + 3)] == "*" or phrase[(tok_index + 3)] == "/":
           if identifiers:
               if phrase[(tok_index + 5)] == ";":
                   print(f"\t\t<Statement> : {tok} {phrase[(tok_index + 1)]}
{phrase[(tok_index + 2)]} {phrase[(tok_index + 3)]} {phrase[(tok_index + 4)]}
{phrase[(tok_index + 5)]}")
           elif integer:
               if phrase[(tok_index + 5)] == ";":
                   print(f"\t\t<Statement> : {tok} {phrase[(tok_index + 1)]}
{phrase[(tok_index + 2)]} {phrase[(tok_index + 3)]} {phrase[(tok_index + 4)]}
{phrase[(tok_index + 5)]}")
           else:
               print(f"\t\t<Statement> : token expected after {phrase[(tok_index +
3)]}")

   if assign_expr1:
       if phrase[(tok_index + 3)] == ";":
           print(f"\t<Statement> : {tok} {phrase[(tok_index + 1)]} {phrase[(tok_index
+ 2)]} {phrase[(tok_index + 3)]}")
       elif phrase[(tok_index + 3)] == "+" or phrase[(tok_index + 3)] == "-" or
phrase[(tok_index + 3)] == "*" or phrase[(tok_index + 3)] == "/":
           if identifiers:
               if phrase[(tok_index + 5)] == ";":
                   print(f"\t\t<Statement> : {tok} {phrase[(tok_index + 1)]}
{phrase[(tok_index + 2)]} {phrase[(tok_index + 3)]} {phrase[(tok_index + 4)]}
{phrase[(tok_index + 5)]}")
           elif integer:
```

```
            if phrase[(tok_index + 5)] == ";":
                print(f"\t\t<Statement> : {tok} {phrase[(tok_index + 1)]}
{phrase[(tok_index + 2)]} {phrase[(tok_index + 3)]} {phrase[(tok_index + 4)]}
{phrase[(tok_index + 5)]}")
        else:
            print(f"\t\t<Statement> : token expected after {phrase[(tok_index +
3)]}")



def control_block(tok, tok_index):
   inv_id = re.search(r'[A-Za-z]([A-Za-z])*.[0-9]([0-9])*', f'{phrase[(tok_index +
2)]}')
   inv_id1 = re.search(r'[A-Za-z]([A-Za-z])*.[0-9]([0-9])*', f'{phrase[(tok_index +
4)]}')


   i = 2
   string = ""
   while i < 5:
       integer =  re.search(r'[0-9]([0-9])*', f'{phrase[(tok_index + i)]}')
       op = re.search(r'==|!=|>|>=|<|<=', f'{phrase[(tok_index + i)]}')
       identifiers = re.search(r'[A-Za-z]([A-Za-z]|[0-9])*', f'{phrase[(tok_index +
i)]}')
       if phrase[(tok_index + 1)] == "(":

           if identifiers:
               pass
           if op:
               pass
           if integer:
               pass
       i += 1
   string += "("
   string += "expression"
   if phrase[(tok_index + 5)] == ")":
       string += ")"
       if phrase[(tok_index + 6)] == "{":
           string += "{"
           i = 7
           while i < len(phrase):
               if phrase[(tok_index + i)] == "{":
```

```
                    i += 1
                    while i < len(phrase):
                        if phrase[(tok_index + i)] == "}":
                            break
                        i += 1
                elif phrase[(tok_index + i)] == "}":
                    # print(f"matching bracket found at line {tok_index + i}")
                    string += "statements }"
                    break
                i += 1
            print(f"\t<control-stmt-{tok}> : {string}")
            print(f'\t\t<Expression> : {phrase[(tok_index + 2)]} {phrase[(tok_index +
3)]} {phrase[(tok_index + 4)]}')
            if inv_id:
                print(f"\t\t\t{inv_id[0]}\t-->\tinvalid identifier!!!")
                return
            if inv_id1:
                print(f"\t\t\t{inv_id1[0]}\t-->\tinvalid identifier!!!")
                return



def parser():
    i = 0
    tmplist = phrase
    for token in phrase:
        control_stmt = re.search(r'if|while', f'{token}')
        for key, value in grammar.items():
            if value in token:
                if token == ("int"):
                    prog(value, phrase[(i+1)])
                    declarations(value, phrase[(i+1)])
                if token == ("float"):
                    declarations(value, phrase[(i+1)])
                if control_stmt:
                    control_block(value, i)
                if token == "return":
                    if phrase[(i + 1)] == "0":
                        if phrase[(i + 2)] == ";":
                            print("\t<return-stmt>")
                        else:
                            print("';' expected")
```

```
                    else:
                        print("token expected after return")


        if i < (len(phrase)-7):
            assignment(token, i)
        i += 1

parser()
```

1.6 parser output for token stream generated (including screenshots of parser output)

With Sample-code.

```
[abelbenard@abel compiler_constr % python3 parser.py
 <Program> : int main () { declarations statements }
        <Declaration> : int x ;
        <Declaration> : int y ;
        <Declaration> : int i =0;
                <Statement> : i = 0 ;
        <Declaration> : int 1a ;
                1a      -->     identifier expected : identifiers can not start with a number
        <Declaration> : float result ;
        <control-stmt-if> : (expression){statements }
                <Expression> : x < y
                <Statement> : result = x + y ;
        <control-stmt-if> : (expression){statements }
                <Expression> : x > y
                <Statement> : result = x - y ;
        <return-stmt>
 abelbenard@abel compiler_constr %
```