Programming Language and Compiler
Final Project Report

B Nischal (st119982)
C Kavin (st120089)
M Sriram (st120001)

Abstract

This report covers the project on Compiler. The compiler is designed for a specific language with given constraints. In this report, a brief description about grammar, semantic rules, type checking and type binding is explained. Screenshots of the demonstration of all the features are shown along with the inputs and outputs given. In the end, it is concluded with the program features and its limitations.

Contents

1 Syntax

1.1 primitives decleration

```
datatype variablename = expression;
```

1.2 primitives assignment

```
variablename = expression;
```

1.3 array decleration

```
array:arraylength variablename = [expression0, expression1];
```

1.4 array member assignment

```
variablename[expression] = expression;
```

1.5 struct decleration

```
struct variablename = {key0: expression0, key1; expression1};
```

1.6 struct member assignment

```
variablename->key =  expression;
```

1.7 while loop

```
while booleanExpression
begin
...
...
end;
```

1.8 function decleration

```
def functionname(type0 argument0, type1 argument1)
begin
....
....
end;
```

1.9 function call

```
functionname(expression0, expression1);
```

2 Grammar and Symantic Rules

```
program         ->  statement_list:s {program.val = statement_list.val}


statement_list  ->  statement_list:l statement_part:s
                    {statement_list = statement_list.add(statement_part)}
                |   statement_part:s
                    {statement_list.add(statement_part)}


statement_part  ->  statement:s SEMI;


statement       ->  assignment:s {statement.val = assignment.val}
                |   vardec:s {statement.val = vardec.val}
                |   RETURN boolExp:e {statement.val = boolExp.val}
                |   ifthen:s {assignment.val statement.val = ifthen.val}
                |   print:s {statement.val = print.val}
                |   while:s {statement.val = while.val}
                |   functionDef:f {statement.val = functionDef.val }
                |   boolExp:e {statement.val = boolExp.val}
                |   BEGIN statement_list:s END {statement.val = statement_list.val}


while           ->  WHILE boolExp:e DO statement:s
                    {while.val = while boolExp.val==true do statement.val}


print           ->  PRINT boolExp:e {print = print boolExp.val}
```

```
ifthen          ->  IF boolExp:e THEN statement
                    {ifthen.val = if boolExp.val==true then statement.val}
                |   IF boolExp:e THEN statement1 ELSE statement2
                    {ifthen.val = if boolExp.val==true
                                then statement1.val else statement2.val}


expListPart     ->  boolExp:e {expListPart.val = boolExp.val}


expList         ->  expList:e_list COMMA expListPart:e
                    {expList.val = expList.add(expListPart.val)}
                |   expListPart:e {expList.add(expListPart.val)}


vardec          ->  type:t ID:i ASS boolExp:e {symboltable.add(t,i,e)}
                |   ARRAYDEF:a COLON boolExp:size ID:i ASS boolExp:e
                    {symboltable.add(a,i,e)}
                |   STRUCTDEF:s ID:i ASS boolExp:e {symboltable.add(s,i,e)}


argumentPart    ->  type:t ID:i  {argumentPart.val = i.val}


argumentList    ->  argumentList:a_list COMMA argumentPart:a
                    {argumentList.val = argumentList.add(argumentPart.val)}
                |   argumentPart:a {argumentList.val = argumentPart.val}


functionDef     ->  FUNCDEF:f ID:i LPAREN argumentList:a_list RPAREN
                        BEGIN statement_list:s END
                    {symboltable.add(f,i,argumentList.val, statement_list:s)}


functionCall    ->  ID:i LPAREN expList:e_list RPAREN
                    {symboltable.get(i).call(expList:e_List.val)}


type            ->  INTDEF:i {type.val = int.val}
                |   FLOATDEF:f {type.val = float.val}
                |   BOOLEANDEF:b {type.val = boolean.val}
                |   CHARDEF:c {type.val = char.val}


keyValuePart    ->  ID:i COLON boolExp:e {keyValuePart.val = boolExp.val}
```

```
keyValueList    ->  keyValueList:kv_list COMMA keyValuePart:kv_part
                    {keyValueList.val = keyValueList.add(keyValuePart.val)}
                |   keyValuePart:kv_part {keyValueList.add(keyValuePart.val)}


assignment      ->  ID:i ASS boolExp:e {symboltable.setValue(i.val, boolExp.val) }
                |   ID:a LBRACKET boolExp:index RBRACKET ASS boolExp:e
                    {symboltable.get(i).get(index) = e.val}
                |   ID:s ARROW ID:key ASS boolExp:e {symboltable.get(i).get(key) = e.val}


boolExp         ->  boolExp:e OR boolTerm:t {boolExp.val = e.val || t.val}
                |   boolTerm:t {boolExp.val = t.val}


boolTerm        ->  boolTerm:t AND notFactor:f {boolTerm.val = t.val && f.val}
                |   notFactor:f {boolTerm.val = f.val}


notFactor       ->  NOT boolFactor:b {notFacto.val = !b.val}
                |   boolFactor:f {notFactor.val = f.val}


boolFactor      ->  BOOLEAN:b {boolFactor.val = b.val}
                |   relation:r {boolFactor.val = r.val}


relation        ->  expr:e0 EQ expr:e1 {relation.val = e0 == e1}
                |   expr:e0 NOTEQ expr:e1 {relation.val = e0 != e1}
                |   expr:e0 GREATEREQ expr:e1 {relation.val = e0 > e1 || e0 == e1}
                |   expr:e0 GREATER expr:e1 {relation.val = e0 > e1}
                |   expr:e0 LESSEREQ expr:e1 {relation.val = e0 < e1 || e0 ==e1}
                |   expr:e0 LESSER expr:e1 {relation.val = e0 < e1}
                |   expr:e {relation.val == e.val}


expr            ->  expr:e PLUS term:t {expr.val = e.val + t.val}
                |   expr:e MINUS term:t {expr.val = e.val - t.val}
                |   term:t {expr.val = t.val}


term            ->  term:t TIMES factor:f {term.val = t.val * f.val}
                |   term:t DIVIDE factor:f {term.val + t.val / f.val}
                |   factor:f { term.val = f.val}
```

```
factor          ->  LPAREN boolExp RPAREN {factor.val = boolExp.val}
                |   INT {factor.val = INT.val}
                |   FLOAT {factor.val = FLOAT.val}
                |   CHAR {factor.val = CHAR.val}
                |   array {factor.val = array.val}
                |   struct {factor.val = struct.val}
                |   functionCall {factor.val = functionCall.val}
                |   ID LBRACKET boolExp RBRACKET
                    {factor.val = symboltable.get(ID).get(boolExp)}
                |   ID:i ARROW ID:j {factor.val = symboltable.get(i).get(j)}
                |   ID {factor.val = ID.val}


array           -> LBRACKET expList:e_list RBRACKET {array.val = expList.val}


struct          -> LBRACE keyValueList:kv_list RBRACE {struct.val = keyValueList.val}
```
‘‘‘

3 Program Features

- This program helps in evaluating expressions like addition(+), subtraction(-), multiplication(*) and division(/). Apart from just mathematical operations, this program also evaluates comparison operators like less than, less than equal, greater than, greater than equal, equality and inequality and logical operators like Conjunction (and), Disjunction (or), Negation (not).
- The program supports the primitive data types Integer(int), Floating point numbers (float), Characters (char) and Booleans (boolean).
- The program supports the composite data types Arrays (array) and Cartesian products (struct).
- The program supports functions with dynamic binding and copying mechanism for parameter passing.

Other Features include:

- Print statement
- Variable declaration statement
- Assignment statement
- While loop

- Conditional statement (If-Else)
- Function declaration and Function call
- Cartesian products
- Arrays
- Type checking
- Boolean cannot operate with any other types and error handling is expected.
- Integer and float may operate with each other. Type conversion mechanism is provided if the operations between integer and float is possible.
- Statement block and environment checking
- The variable declared within a child environment should not be used in any higher level environment.

4 Limitations

- The program can not handle multi-dimensional arrays.

5 Member Responsibilty

- Grammar Rules Production – Done by all of us.
- Type checking – Sriram and Nischal
- Report – Kavin
- *.java files – Nischal

6 Examples

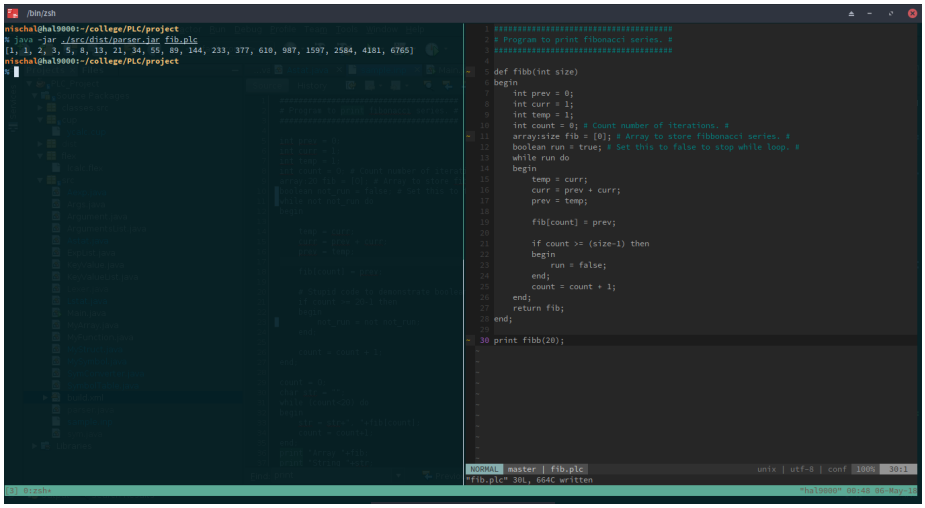

Figure 1: program to generate fibonacci series

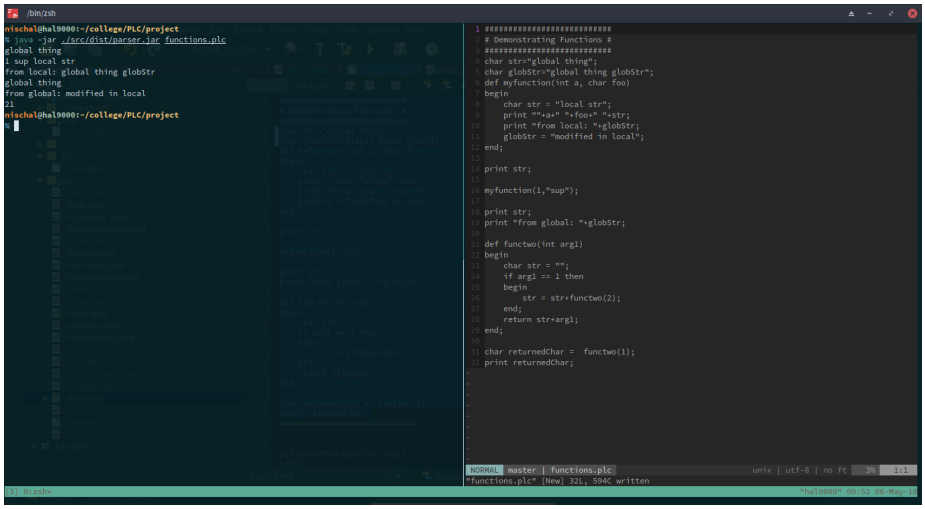

Figure 2: demonstrating functions

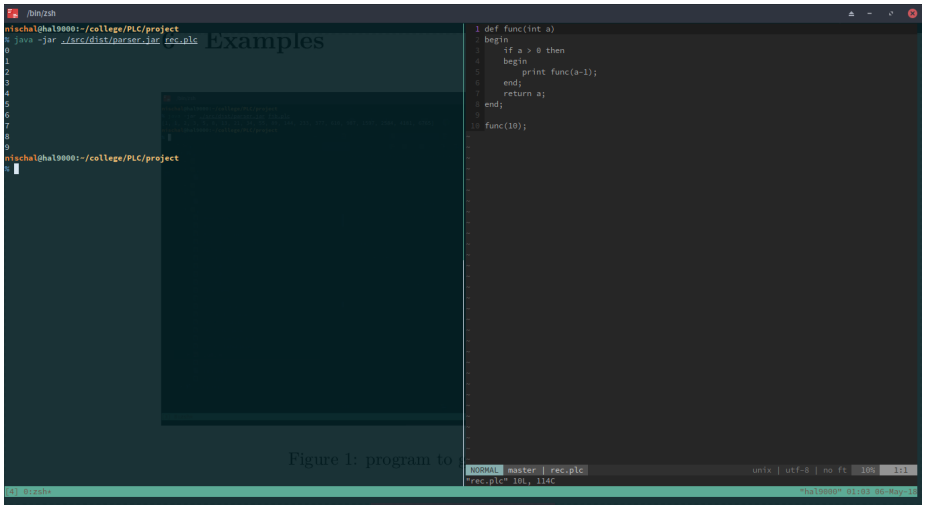

Figure 3: demonstrating recursive functions

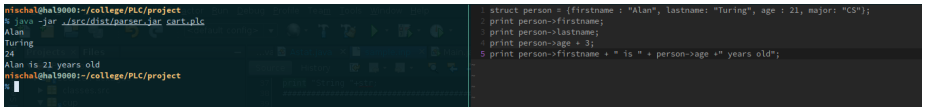

Figure 4: demonstrating cartesian product