Department of Computer Science and Engineering, Section A
National Institute of Technology, Tiruchirappalli

CSPC62 - Compiler Design

Designing a Compiler for Base Typescript

Dhrubit Hajong (106121037)
Mercia Melvin Perinchery (106121077)
Nishith Eedula (106121085)

Index

Phase 1 : Lexical Analyser

I. Components of the Language - Base TypeScript

TypeScript is a free and open-source high-level programming language developed by Microsoft that adds static typing with optional type annotations to JavaScript. It is primarily designed to develop large-scale web applications due to its enhanced data integrity and error handling.

Base TypeScript is a significant subset of the TypeScript programming language comprising the essential features of TypeScript. The components of our language are as follows.

a. Keywords

Keywords are reserved words with special meaning to the compiler that cannot be used as identifiers. Some important keywords we have defined in our language are as follows.

```
import  from  let  const  var  console
```

b. Symbols

Symbols or identifiers define the names of the language's components. In Typescript, an identifier can contain a sequence of letters, digits, underscores and dollar symbols, provided it does not start with a digit.

```
identifier
```

c. Data Types

A data type is a property of a language component that lets the compiler know how to interpret its value. Apart from the usual types, we have also defined some additional utility types from Typescript.

```
any  number  string  boolean  undefined  type
```

d. Functions

A function is an organised set of instructions used to repeatedly perform tasks in the program. Some functions and keywords we have defined in our language are as follows.

```
function  async  =>  console.log()
```

e. Control Structures

A control structure is used to choose the direction of flow of the program. Some important control structures and keywords we have defined in our language are as follows.

```
if  else  for  while  return  break  continue
```

f. Operators

Operators are symbols that tell the compiler to perform a specific mathematical, relational or logical operation on operands to produce a final result. Some of the important operators defined in our language are as follows.

```
&&  ||  !  =  ==  !=  <  <=  >  >=  +  -  *  /
```

g. Punctuation

Punctuators are symbols that have syntactic and semantic meaning to the compiler but do not specify an operation that yields a value as such. Some of the important punctuation defined in our language are as follows.

```
;  :  ,  .
```

h. Comments

Comments are Human/Programmer Readable Descriptions that detail the intent or purpose of the code. Used for documentation and resource inclusion, the compiler considers them as non-executable statements.
in our language are denoted by

```
//  /*  */
```

i. Other Symbols

Some other utility symbols defined in our language are as follows.

```
(  )  {  }  &  |
```

II. Constructing the Deterministic Finite Automata for the Regular Expressions

Integer

Float

Boolean

String

Data Type

Identifier

Comments

III. FLEX Code for Implementing Patterns and Actions

```
%{
    #include <bits/stdc++.h>
    #include "colors.h"
    using namespace std;

    #define TOKEN(t) TokenType(t, string(yytext, yyleng))
    #define INCREMENT loc.col += yyleng
    #define ERRORMESSAGE(e) cerr << BOLDRED << "Line : " << loc.line << ", Col : "
    << loc.col << BOLDRED << "\t" << e << " \"" << BOLDRED << string(yytext,yyleng)
    << "\"" << RESET << endl

    extern FILE* yyin;
    struct LocationType {
        int col = 0;
        int line = 0;
    };
    LocationType loc;

    /* Tokens */
    enum TokenNames {
        IMPORT, FROM,
        LET, CONST, VAR,
        FUNCTION, ASYNC, ARROW,
        IF, ELSE, FOR, WHILE,
        RETURN, BREAK, CONTINUE,
        INTEGER, FLOAT, BOOLEAN, STRINGVALUE,
        ANYTYPE, NUMBERTYPE, STRINGTYPE, BOOLEANTYPE, UNDEFINED,
        TYPE, DATATYPE, IDENTIFIER,
        LEFTROUND, RIGHTROUND, LEFTCURLY, RIGHTCURLY,
        SEMICOLON, COLON, COMMA, PERIOD,
        AND, OR, NOT,
        UNION, INTERSECTION,
        EQUAL, DOUBLEEQUAL, NOTEQUAL,
        LESSTHAN, LESSEQUAL, GREATERTHAN, GREATEREQUAL,
        PLUS, MINUS, MULTIPLY, DIVIDE,
        CONSOLELOG,
        SINGLECOMMENT, MULTICOMMENT,
    };

    struct TokenType {
        TokenNames tokenName;
        string literal;
        LocationType locationData;
        TokenType(TokenNames tokName, string lit): tokenName(tokName), literal(lit),
        locationData(loc){}
    };

    vector<TokenType> tokenList;
    unordered_map<TokenNames,string> tokenMap = {
        {IMPORT, "IMPORT"}, {FROM, "FROM"},
        {LET, "LET"}, {CONST, "CONST"}, {VAR, "VAR"},
        {FUNCTION, "FUNCTION"}, {ASYNC, "ASYNC"}, {ARROW, "ARROW"},
        {IF, "IF"}, {ELSE, "ELSE"}, {FOR, "FOR"}, {WHILE, "WHILE"},
        {RETURN, "RETURN"}, {BREAK, "BREAK"}, {CONTINUE, "CONTINUE"},
        {INTEGER, "INTEGER"}, {FLOAT, "FLOAT"}, {BOOLEAN, "BOOLEAN"},
        {STRINGVALUE, "STRINGVALUE"},
        {ANYTYPE, "ANYTYPE"}, {NUMBERTYPE, "NUMBERTYPE"}, {STRINGTYPE, "STRINGTYPE"},
        {BOOLEANTYPE, "BOOLEANTYPE"}, {UNDEFINED, "UNDEFINED"},
        {TYPE, "TYPE"}, {DATATYPE, "DATATYPE"}, {IDENTIFIER, "IDENTIFIER"},
        {LEFTROUND, "LEFTROUND"}, {RIGHTROUND, "RIGHTROUND"},
        {LEFTCURLY, "LEFTCURLY"}, {RIGHTCURLY, "RIGHTCURLY"},
        {SEMICOLON, "SEMICOLON"}, {COLON, "COLON"}, {COMMA, "COMMA"},
        {PERIOD, "PERIOD"}, {AND, "AND"}, {OR, "OR"}, {NOT, "NOT"},
        {UNION, "UNION"}, {INTERSECTION, "INTERSECTION"},
        {EQUAL, "EQUAL"}, {DOUBLEEQUAL, "DOUBLEEQUAL"}, {NOTEQUAL, "NOTEQUAL"},
        {LESSTHAN, "LESSTHAN"}, {LESSEQUAL, "LESSEQUAL"},
        {GREATERTHAN, "GREATERTHAN"}, {GREATEREQUAL, "GREATEREQUAL"},
        {PLUS, "PLUS"}, {MINUS, "MINUS"}, {MULTIPLY, "MULTIPLY"}, {DIVIDE, "DIVIDE"},
        {CONSOLELOG, "CONSOLELOG"},
        {SINGLECOMMENT, "SINGLECOMMENT"}, {MULTICOMMENT, "MULTICOMMENT"},
    };
    extern "C" int yywrap() { return 1; }
    bool errorFlag = false;
%}

/* Rules */
%%
[ \t]                       { loc.col++; }
[\n]                        { loc.line++; loc.col = 0; }
"import"                    { INCREMENT; tokenList.push_back(TOKEN(IMPORT)); }
"from"                      { INCREMENT; tokenList.push_back(TOKEN(FROM)); }
"let"                       { INCREMENT; tokenList.push_back(TOKEN(LET)); }
"const"                     { INCREMENT; tokenList.push_back(TOKEN(CONST)); }
"var"                       { INCREMENT; tokenList.push_back(TOKEN(VAR)); }
"function"                  { INCREMENT; tokenList.push_back(TOKEN(FUNCTION)); }
"async"                     { INCREMENT; tokenList.push_back(TOKEN(ASYNC)); }
"=>"                        { INCREMENT; tokenList.push_back(TOKEN(ARROW)); }
"if"                        { INCREMENT; tokenList.push_back(TOKEN(IF)); }
"else"                      { INCREMENT; tokenList.push_back(TOKEN(ELSE)); }
"for"                       { INCREMENT; tokenList.push_back(TOKEN(FOR)); }
"while"                     { INCREMENT; tokenList.push_back(TOKEN(WHILE)); }
"return"                    { INCREMENT; tokenList.push_back(TOKEN(RETURN)); }
"break"                     { INCREMENT; tokenList.push_back(TOKEN(BREAK)); }
"continue"                  { INCREMENT; tokenList.push_back(TOKEN(CONTINUE)); }
[0-9_]+                     { INCREMENT; tokenList.push_back(TOKEN(INTEGER)); }
[0-9_]+\.[0-9_]*            { INCREMENT; tokenList.push_back(TOKEN(FLOAT)); }
("true"|"false")            { INCREMENT; tokenList.push_back(TOKEN(BOOLEAN)); }
["].*["]                    { INCREMENT; tokenList.push_back(TOKEN(STRINGVALUE)); }
"any"                       { INCREMENT; tokenList.push_back(TOKEN(ANYTYPE)); }
"number"                    { INCREMENT; tokenList.push_back(TOKEN(NUMBERTYPE)); }
"string"                    { INCREMENT; tokenList.push_back(TOKEN(STRINGTYPE)); }
"boolean"                   { INCREMENT; tokenList.push_back(TOKEN(BOOLEANTYPE)); }
"undefined"                 { INCREMENT; tokenList.push_back(TOKEN(UNDEFINED)); }
"type"                      { INCREMENT; tokenList.push_back(TOKEN(TYPE)); }
```

```
[:][ ]*[a-z]+               { INCREMENT; tokenList.push_back(TOKEN(DATATYPE)); }
[_$a-zA-Z][_a-zA-Z0-9$]*   { INCREMENT; tokenList.push_back(TOKEN(IDENTIFIER)); }
"("                         { INCREMENT; tokenList.push_back(TOKEN(LEFTROUND)); }
")"                         { INCREMENT; tokenList.push_back(TOKEN(RIGHTROUND)); }
"{"                         { INCREMENT; tokenList.push_back(TOKEN(LEFTCURLY)); }
"}"                         { INCREMENT; tokenList.push_back(TOKEN(RIGHTCURLY)); }
";"                         { INCREMENT; tokenList.push_back(TOKEN(SEMICOLON)); }
":"                         { INCREMENT; tokenList.push_back(TOKEN(COLON)); }
","                         { INCREMENT; tokenList.push_back(TOKEN(COMMA)); }
"."                         { INCREMENT; tokenList.push_back(TOKEN(PERIOD)); }
"&&"                        { INCREMENT; tokenList.push_back(TOKEN(AND)); }
"||"                        { INCREMENT; tokenList.push_back(TOKEN(OR)); }
"!"                         { INCREMENT; tokenList.push_back(TOKEN(NOT)); }
"|"                         { INCREMENT; tokenList.push_back(TOKEN(UNION)); }
"&"                         { INCREMENT; tokenList.push_back(TOKEN(INTERSECTION)); }
"="                         { INCREMENT; tokenList.push_back(TOKEN(EQUAL)); }
"=="                        { INCREMENT; tokenList.push_back(TOKEN(DOUBLEEQUAL)); }
"!="                        { INCREMENT; tokenList.push_back(TOKEN(NOTEQUAL)); }
"<"                         { INCREMENT; tokenList.push_back(TOKEN(LESSTHAN)); }
"<="                        { INCREMENT; tokenList.push_back(TOKEN(LESSEQUAL)); }
">"                         { INCREMENT; tokenList.push_back(TOKEN(GREATERTHAN)); }
">="                        { INCREMENT; tokenList.push_back(TOKEN(GREATEREQUAL)); }
"+"                         { INCREMENT; tokenList.push_back(TOKEN(PLUS)); }
"-"                         { INCREMENT; tokenList.push_back(TOKEN(MINUS)); }
"*"                         { INCREMENT; tokenList.push_back(TOKEN(MULTIPLY)); }
"/"                         { INCREMENT; tokenList.push_back(TOKEN(DIVIDE)); }
"console.log(".*")"         { INCREMENT; tokenList.push_back(TOKEN(CONSOLELOG)); }
[/][/].*                    { INCREMENT; tokenList.push_back(TOKEN(SINGLECOMMENT)); }
[/][*](.|\n)*.*[*][/]       { INCREMENT; tokenList.push_back(TOKEN(MULTICOMMENT)); }
.                    { ERRORMESSAGE("Unrecognised Symbol"); errorFlag = true;
                     yyterminate(); }
[0-9]+[_a-zA-Z$]*    { ERRORMESSAGE("Invalid Identifier"); errorFlag = true;
                     yyterminate(); }
%%

/* Main Function */
int main() {
	yyin = fopen("input.ts", "r");
	yylex();
	if(errorFlag) {
		cerr << BOLDRED << UNDERLINE << "Error generated" << RESET << endl;
		return 1;
	}
	bool displayLines = true;
	for(auto &x: tokenList){
		if(displayLines){
			cout << " Line: " << BOLDBLUE << x.locationData.line << RESET
			 <<  ", Col: " << BOLDBLUE << x.locationData.col << RESET << "\t";
		}
		cout << BOLDMAGENTA << x.literal << "\t\t\t" << BOLDGREEN
		<< tokenMap.at(x.tokenName) << RESET << "\n";
	}
}
```

IV. Sample Input Program and Terminal Output

Input Program (TypeScript File)

```
// Ignored
/* Ignored */

let valueString: string = "Dru";
let valueNumber: number = 789;
let valueFloat: number = 0.123_456;

type Employee = {
    id: number;
    name: string;
};

type Details = Employee | undefined;

function checkEmployee(emp: Employee) {
    if (emp.id && emp.name) {
        return true;
    } else {
        return false;
    }
}

console.log(valueNumber);
```

Terminal Output

```
Line: 0, Col: 10       // Ignored              SINGLECOMMENT
Line: 1, Col: 13       /* Ignored */           MULTICOMMENT
Line: 3, Col: 3        let                     LET
Line: 3, Col: 15       valueString             IDENTIFIER
Line: 3, Col: 23       : string                DATATYPE
Line: 3, Col: 25       =                       EQUAL
Line: 3, Col: 31       "Dru"                   STRINGVALUE
Line: 3, Col: 32       ;                       SEMICOLON
Line: 4, Col: 3        let                     LET
Line: 4, Col: 15       valueNumber             IDENTIFIER
Line: 4, Col: 23       : number                DATATYPE
Line: 4, Col: 25       =                       EQUAL
Line: 4, Col: 29       789                     INTEGER
Line: 4, Col: 30       ;                       SEMICOLON
Line: 5, Col: 3        let                     LET
Line: 5, Col: 14       valueFloat              IDENTIFIER
Line: 5, Col: 22       : number                DATATYPE
Line: 5, Col: 24       =                       EQUAL
Line: 5, Col: 34       0.123_456               FLOAT
Line: 5, Col: 35       ;                       SEMICOLON
Line: 7, Col: 4        type                    TYPE
Line: 7, Col: 13       Employee                IDENTIFIER
Line: 7, Col: 15       =                       EQUAL
```

```
Line: 7, Col: 17        {                       LEFTCURLY
Line: 8, Col: 6         id                      IDENTIFIER
Line: 8, Col: 14        : number                DATATYPE
Line: 8, Col: 15        ;                       SEMICOLON
Line: 9, Col: 8         name                    IDENTIFIER
Line: 9, Col: 16        : string                DATATYPE
Line: 9, Col: 17        ;                       SEMICOLON
Line: 10, Col: 1        }                       RIGHTCURLY
Line: 10, Col: 2        ;                       SEMICOLON
Line: 12, Col: 4        type                    TYPE
Line: 12, Col: 12       Details                 IDENTIFIER
Line: 12, Col: 14       =                       EQUAL
Line: 12, Col: 23       Employee                IDENTIFIER
Line: 12, Col: 25       |                       UNION
Line: 12, Col: 35       undefined               UNDEFINED
Line: 12, Col: 36       ;                       SEMICOLON
Line: 14, Col: 8        function                FUNCTION
Line: 14, Col: 22       checkEmployee           IDENTIFIER
Line: 14, Col: 23       (                       LEFTROUND
Line: 14, Col: 26       emp                     IDENTIFIER
Line: 14, Col: 27       :                       COLON
Line: 14, Col: 36       Employee                IDENTIFIER
Line: 14, Col: 37       )                       RIGHTROUND
Line: 14, Col: 39       {                       LEFTCURLY
Line: 15, Col: 6        if                      IF
Line: 15, Col: 8        (                       LEFTROUND
Line: 15, Col: 11       emp                     IDENTIFIER
Line: 15, Col: 12       .                       PERIOD
Line: 15, Col: 14       id                      IDENTIFIER
Line: 15, Col: 17       &&                      AND
Line: 15, Col: 21       emp                     IDENTIFIER
Line: 15, Col: 22       .                       PERIOD
Line: 15, Col: 26       name                    IDENTIFIER
Line: 15, Col: 27       )                       RIGHTROUND
Line: 15, Col: 29       {                       LEFTCURLY
Line: 16, Col: 14       return                  RETURN
Line: 16, Col: 19       true                    BOOLEAN
Line: 16, Col: 20       ;                       SEMICOLON
Line: 17, Col: 5        }                       RIGHTCURLY
Line: 17, Col: 10       else                    ELSE
Line: 17, Col: 12       {                       LEFTCURLY
Line: 18, Col: 14       return                  RETURN
Line: 18, Col: 20       false                   BOOLEAN
Line: 18, Col: 21       ;                       SEMICOLON
Line: 19, Col: 5        }                       RIGHTCURLY
Line: 20, Col: 1        }                       RIGHTCURLY
Line: 22, Col: 24       console.log(valueNumber)CONSOLELOG
Line: 22, Col: 25       ;                       SEMICOLON
```

V. Error Handling with Terminal Output

Input Program (with Unrecognised Symbols)

```
let %valueString%: string = "Dru";
let valueNumber: number = 789;
let valueFloat: number = 0.123_456;
```

Terminal Output

```
Line : 3, Col : 4       Unrecognised Symbol "%"
Error generated
```

Input Program (with Invalid Identifiers)

```
let 123valueString: string = "Dru";
let valueNumber: number = 789;
let valueFloat: number = 0.123_456;
```

Terminal Output

```
Line : 3, Col : 4       Unrecognised Symbol "123valueString"
Error generated
```