

오토마타와 컴파일러

Final Report: 계산기 프로그램(인터프리터)

학번, 이름	32131766 황준일
제출일	2019-12-17
담당 교수	이 상 범 교수님

1. 결과물: <https://junilhwang.github.io/automata-compiler/>

[크롬 브라우저에서 작동하는 계산기 프로그램을 만들었습니다]

A. 정상적 처리

웹 인터프리터

Input	Output
Program testProgram; /* 테스트 프로그램 선언	10
Var	2
int a, b, c; /* 정수 변수 선언	112
float x, y, z; /* 실수 변수 선언	100
int d, e; /* 정수 변수 추가 선언	12
Begin	1
a = 10; b = 2; c = a^b + (a+b);	1.5
print a; print b; print c; print a^b; print a+b;	2.5
x = 1.0; y = 1.5; z = x + y;	
print x; print y; print z;	
/* 프로그램 실행 완료	
End	

과제에서 요구한 내용을 반영한 결과물입니다.

Input영역과 Output 영역을 구분하였습니다.

B. 에러처리

Var	Begin	End	Program
Error : Next state is not Var	Error : Next state is not Begin	Error : Next state is not End	Error: Token Type must be 'VarName(word)'

다음 Program State에 대한 에러처리입니다. Program → Var → Begin → End 순서가 지켜져야 합니다.

Program test; Begin	Program test; End	Program test; Program
Error : Next state is not Begin	Error : Next state is not End	Error: Next state must be 'Var'

마찬가지로 Program State가 지켜져야 합니다.

```
Program test;
Var
a = 10;
```

Error: Assignment is possible at 'Begin'

```
Program test;
Var
int a, a;
```

Error: 'a' is already defined Variable

Var State에서는 변수 할당이 불가능하고, 이미 선언 된 변수를 또 선언할 수 없습니다.

```
Program test;
Var
int b;
Program
```

Error: Next state must be 'Var'

```
Program test;
Var
int b;
Var
```

Error: Next state is not 'Var'

```
Program test;
Var
int b;
End
```

Error: Next state is not 'End'

위와 마찬가지로 Program State에 대한 순서에 대해 체크합니다.

```
Program test;
Var
int b;
Begin
a = 10;
```

Error: 'a' undefined

```
Program test;
Var
int b;
Begin
print a;
```

Error: 'a' is not defined

```
Program test;
Var
int b;
Begin
print b;
```

Error: 'b' is not assign

```
Program test;
Var
int b;
Begin
b = 10
```

Error: Last token must be 'Semicolon(;)'

Var에서 선언하지 않은 변수에 대해서만 할당할 수 있습니다.

그리고 할당하지 않은 변수를 사용하거나 출력할 때 assign을 해야 된다는 에러를 띄웁니다.

```
print 10;
```

Error: "Print" function must use 'Begin' state

```
Program test;
print 10;
```

Error: "Print" function must use 'Begin' state

```
Program test;
Var
print 10;
```

Error: "Print" function must use 'Begin' state

```
Program test;
Var
Begin
End
print 10;
```

Error: "Print" function must use 'Begin' state

Print는 Begin state에서만 사용할 수 있습니다.

2. 소스코드 설명

* 웹 페이지에 결과물을 렌더링하는 부분(Framework 부분)을 제외하고 설명하도록 하겠습니다.

A. CodeContainer.ts

```
class CodeContainer {
  public static instance: CodeContainer; // 싱글톤으로 사용될 instance
  public static init(): void {
    CodeContainer.instance = new CodeContainer(''); // 싱글톤 인스턴스 생성
  }
  public static getInstance(): CodeContainer {
    return CodeContainer.instance; // 싱글톤 인스턴스 반환
  }
  private code: string = '';
  constructor(code: string) {
    this.setCode(code); // code 를 받아와서 저장함
  }
  public setCode(code: string) {
    this.code = code.trim(); // 양 끝 여백을 없앴
  }
  public getCode() {
    return this.code.trim(); // 현재 코드 반환
  }
  public getFirst() {
    return this.getCode()[0]; // 첫 번째 글자 반환
  }
}
CodeContainer.init(); // 클래스로부터 static 상태의 싱글톤 인스턴스 생성

// codeContainer 변수를 만든 후 export
export const codeContainer: CodeContainer = CodeContainer.getInstance();
```

B. TokenDefined.ts (1)

```
// Token 의 interface 정의이 */
export interface Token {
  type: number;
  value?: number|string;
}

// Token Type 정의. counter 변수를 사용하여 반복 할당 작업 자동화 */
let counter: number = 0;
export const Print: number = counter++;
export const Lparen: number = counter++;
export const Rparen: number = counter++;
export const Plus: number = counter++;
export const Minus: number = counter++;
export const Multi: number = counter++;
export const Divi: number = counter++;
export const Power: number = counter++;
export const Assign: number = counter++;
export const IntNum: number = counter++;
export const VarName: number = counter++;
export const Program: number = counter++;
export const Var: number = counter++;
export const Begin: number = counter++;
export const End: number = counter++;
export const TypeInt: number = counter++;
export const TypeFloat: number = counter++;
export const Empty: number = counter++;
export const Comma: number = counter++;
export const Semicolon: number = counter++;
export const Comment: number = counter++;
```

TokenDefined.ts (2)

```
// 다른 코드에서 사용 될 token 의 key 정의
export const printKey = 'print';
export const programKey = 'Program';
export const varKey = 'Var';
export const beginKey = 'Begin';
export const endKey = 'End';
export const commentKey = '/*';
export const opTable: any = {
  '(': Lparen,
  ')': Rparen,
  '+': Plus,
  '-': Minus,
  '*': Multi,
  '/': Divi,
  '^': Power,
  '=': Assign,
  [printKey]: Print,
  [programKey]: Program,
  [varKey]: Var,
  [beginKey]: Begin,
  [endKey]: End,
  'int': TypeInt,
  'float': TypeFloat,
  ',': Comma,
  ';': Semicolon,
  [commentKey]: Comment,
};
```

TokenDefined.ts (3)

```
// symbol table 에 대한 instance 정의
export const symbolTable = new class {
  private symbols: any = {};
  // symbol 을 새로 만듦. 즉, define 할 때 사용
  public set(key: string, value: number|null = null, type = IntNum) {
    const check = this.symbols[key] === undefined;
    if (check) {
      this.symbols[key] = {type, value};
    }
    return check;
  }
  // symbol 에서 value 를 가져옴
  public get(key: string) {
    const k = this.symbols[key];
    console.log(k);
    return k === undefined ? undefined : k.value;
  }
  // symbol 의 내용을 수정함. 즉, assign 할 때 사용
  public put(key: string, value: number) {
    const check = this.symbols[key] !== undefined;
    if (check) {
      this.symbols[key].value = value;
    }
    return check;
  }
}();
```

C. Scanner.ts

```
import {
  Token, opTable, IntNum, VarName, Empty, Comment,
  printKey, programKey, commentKey, TypeInt, TypeFloat,
} from './TokenDefined'; // 사용될 token 정보들을 가져옴
import { codeContainer } from './CodeContainer'; // codeContainer 를 가져옴
import { isNumChar, isVar } from '@/Helper'; // 문자열이 number 인지, variable 인지 체크하는
  helper function 을 가져옴

export const nextToken = (): Token => {
  const code = codeContainer.getCode(); // 남아있는 code
  const first = codeContainer.getFirst(); // 첫 글자를 가져옴
  const last = code.length; // 남아있는 code 의 길이
  let token: Token; // token 에 저장될
  let i = 0; // i 번 까지의 코드를 삭제
  switch (true) {
    case code.indexOf(commentKey) === 0: // 주석 체크
      i = last;
      token = { type: opTable[commentKey] };
      break;
    case opTable[first] !== undefined: // 단일 글자 token 체크
      i = 1;
      token = { type: opTable[first] };
      break;
    case code.indexOf(printKey) === 0: // print keyword 체크
      i = printKey.length;
      token = { type: opTable[printKey] };
      break;
    case code.indexOf(programKey) === 0: // program keyword 체크
      i = programKey.length;
      token = { type: opTable[programKey] };
      break;
    case isNumChar(first): // number check
      while (isNumChar(code.substr(0, i + 1)) && i < last) { i++; }
      token = { type: IntNum, value: +code.substr(0, i) };
      break;
    case code.indexOf('int') === 0: // int keyword 체크
      i = 3;
      token = { type: TypeInt };
      break;
    case code.indexOf('float') === 0: // float keyword 체크
      i = 5;
      token = { type: TypeFloat };
      break;
    case code.length === 0 : // 빈 문자열 체크
      token = { type: Empty };
      break;
    default: // variable 을 가져옴
      while (isVar(code.substr(0, i + 1)) && i < last) { i++; }
      token = { type: VarName, value: code.substr(0, i) };
      break;
  }
  // code container 의 code 를 code[i] ~ code[last]로 교체
  // 예를들어 code -> int a = 10; 일 경우
  // token -> int, i -> 3, code -> a = 10; 으로 저장됨
  codeContainer.setCode(code.substr(i));
  return token;
};
```

D. Parser.ts

```
import {
  Token, Lparen, Rparen, IntNum, VarName, Assign, Plus, Minus, Multi, Divi, Power, Print, symbolTable, varKey,
  endKey, beginKey, Program, TypeInt, TypeFloat, Empty, Comma, Semicolon, Comment,
} from './TokenDefined'; // TokenType, TokenKey, SymbolTable 등을 가져옴
import { nextToken } from './Scanner'; // scanner 함수 가져옴
import { eventBus } from './Helper'; // 렌더링 시스템에 보낼 함수(이벤트)
import { codeContainer } from './CodeContainer'; // CodeContainer를 가져옴

let stateCounter = 0; // Program State Counter
const NEXT_PROGRAM = stateCounter++;
const NEXT_VAR = stateCounter++;
const NEXT_BEGIN = stateCounter++;
const NEXT_END = stateCounter++;

// 시작은 Program 부터. 즉, 다음 상태로 Program 이 선언되어야함
let parserState: number = NEXT_PROGRAM;

// 다음 state 에 관한 정보. Program->Var, Var->Begin, Begin->End
const nextStateChecker: any = {
  [varKey]: NEXT_VAR,
  [beginKey]: NEXT_BEGIN,
  [endKey]: NEXT_END,
};

let token: Token; // scanner 로 읽어온 token 을 정보를 저장할 변수
const stack: any = []; // stack 정보

const expression = () => {
  // 곱하기, 나누기 처리 후
  term();

  // 더하기, 빼기를 처리
  while ([Plus, Minus].indexOf(token.type) !== -1) {
    operateCallee(term);
  }
};

const term = () => {
  // 제곱근 처리 후
  pow();

  // 곱하기 나누기 처리
  while ([Multi, Divi].indexOf(token.type) !== -1) {
    operateCallee(pow);
  }
};

const pow = () => {
  factor(); // 변수할당, 괄호 연산자, 선언자 처리 후
  while (token.type === Power) {
    operateCallee(factor); // 제곱근 처리
  }
};

// operate 과정 추상화
const operateCallee = (callback: any) => {
  const operator = token.type;
  token = nextToken();
  callback();
  operate(operator);
};
```

```

// 연산 결과를 stack 에 저장함.
const factor = () => {
  const { type, value } = token;
  switch (type) {
    // 변수의 값을 stack 에 저장하는 과정
    case VarName:
      // SymbolTable 에서 변수의 값을 가져온다.
      const symbolValue = symbolTable.get(String(value));

      // 사용하려는 변수가 정의 되지 않았을 때
      if (symbolValue === undefined) {
        errorMessageAppend(`Error: '${value}' is not defined`);
      }

      // 사용하려는 변수가 할당 되지 않았을 때
      if (symbolValue === null) {
        errorMessageAppend(`Error: '${value}' is not assign`);
      }

      // 정상적이라면 stack 에 저장
      stack.push(symbolValue);
      break;

    // 숫자는 바로 stack 에 저장
    case IntNum:
      stack.push(value);
      break;

    // 왼쪽 괄호가 나왔을 경우엔
    case Lparen:
      // 괄호 사이에 수식 처리 후
      token = nextToken();
      expression();
      // 오른쪽 괄호가 나왔는지 검사
      checkTokenType(Rparen, 'Rparen Error');
      break;
  }

  // 다음 토큰 가져오기
  token = nextToken();
};

// 스택에서 값을 가져온 후 더하기, 빼기, 곱하기, 나누기, 제곱 연산 수행
const operate = (operator: number) => {
  const [d2, d1] = [stack.pop(), stack.pop()];
  if (d1 === undefined) { return; }
  switch (operator) {
    case Plus: stack.push(d1 + d2); break;
    case Minus: stack.push(d1 - d2); break;
    case Multi: stack.push(d1 * d2); break;
    case Divi: stack.push(d1 / d2); break;
    case Power: stack.push(Math.pow(d1, d2)); break;
  }
};

// 변수 선언 처리
const defineVariable = (type: number) => {
  do {
    // 토큰을 가져온 후
    token = nextToken();

    // 변수 이름 형태인지 검사
    checkTokenType(VarName, 'Error: Next token must be \'VarName\'');

    // 이미 변수가 선언 되었을 경우 에러처리

```



```

if (!symbolTable.set(String(token.value), null, type)) {
    errorMessageAppend(`Error: '${token.value}' is already defined Variable`);
}

// 다음 토큰이 comma 일 경우 위의 과정 다시 반복
token = nextToken();
} while (token.type === Comma);
};

// 토큰 타입 검사
const checkTokenType = (tokenType: number, message: string): void => {
    const checked = token.type === tokenType;
    if (!checked) {
        // 원하는 토큰 타입이 아닐 경우 에러 호출
        errorMessageAppend(message);
    }
};

// 위의 과정을 통합한, statement 처리 과정
export const statement = (): void => {
    token = nextToken();
    let indent = '\t'; // statement 가 정상이라면 앞에 indent(tab)를 붙여서 출력시킴
    do {
        // 현재 token 의 type 과 value 저장
        const { type: startTokenType, value: startTokenValue } = token;

        // Statement 의 시작(첫번째) token type 에 따라 다른 과정 처리
        switch (startTokenType) {
            // Program 일 경우
            case Program:
                // 현재 parserState 가 NEXT_PROGRAM 이 아니라면 에러 처리
                if (parserState !== NEXT_PROGRAM) {
                    errorMessageAppend('Error: Next state must be \'Var\'');
                }
                token = nextToken();

                // Program 다음엔 변수 이름 형태가 와야됨
                checkTokenType(VarName, 'Error: Token Type must be \'VarName(word)\'');

                // 다음 State 는 Var
                parserState = NEXT_VAR;
                symbolTable.set(String(token.value));
                token = nextToken();
                indent = ''; // Program state 앞에는 indent 가 필요 없음
                break;

            // 출력을 처리하는 부분
            case Print:
                // Print 는 Begin State 에서만 사용할 수 있음
                if (parserState !== NEXT_END) {
                    errorMessageAppend('Error: "Print" function must use \'Begin\' state');
                }
                token = nextToken();
                expression(); // Print 다음에 오는 token 들을 연산 후
                outputAppend(stack.pop()); // Output 영역에 출력함
                break;

            // Statement 시작이 변수 이름일 경우엔 할당문이 필요함
            case VarName:
                // 할당문은 Begin State 에서만 사용할 수 있음
                if (parserState !== NEXT_END) {
                    errorMessageAppend('Error: Assignment is possible at \'Begin\'');
                }
        }
    }

```

```

// 할당문인지 검사
token = nextToken();
checkTokenType(Assign, 'Error: Token Type must be \'Assign(=)\'');

// 할당 처리
token = nextToken();
expression();

// 선언 되지 않은 변수일 경우 예러 처리
if (!symbolTable.put(String(startTokenValue), stack.pop())) {
    errorMessageAppend('Error: '${startTokenValue}' undefined');
}
break;

// int, float 변수 선언 처리
case TypeInt:
case TypeFloat:
    // 변수 선언은 Var state에서만 가능함
    if (parserState !== NEXT_BEGIN) {
        errorMessageAppend('Error: Definition is possible at \'Var\''');
    }
    // 선언 과정은 definedVariable 함수에 위임함
    defineVariable(startTokenType);
    break;
}

// Statement 마지막은 세미콜론이나 주석으로 끝나야함
if ([Semicolon, Comment].indexOf(token.type) === -1) {
    errorMessageAppend('Error: Last token must be \'Semicolon(;)\'');
}

// 다음 토큰이 비어있지 않다면, 즉, 세미콜론 이후에 내용이 있다면 위의 과정 반복
token = nextToken();
} while (token.type !== Empty);

// 한 line 이 정상적으로 처리되었다면 input 영역에 indent 를 추가하여 출력
lineAppend(indent);
};

// Statement 를 처리하기 전, 제일 먼저 state 를 처리해야함
export const checkState = (): void => {
    const code = codeContainer.getCode();
    const next = nextStateChecker[code]; // Var, Begin, End 중 하나를 가져옴
    const checked: boolean = next !== undefined;

    // 한 line 의 코드가 Var, Begin, End 등과 일치하지 않을 때 statement 를 호출함
    if (checked) {
        if (next === parserState) { // 현재 state 와 next 가 같다면
            // state 1 증가. 즉, Var->Begin, Begin->End, End->Program 으로 변경됨
            parserState = (parserState + 1) % stateCounter;
            // 현재 line(code)를 input 영역에 추가
            lineAppend(); return;
        } else {
            // 일치하지 않으면 예러
            errorMessageAppend('Error : Next state is not ${code}`');
        }
    }
    statement();
};

// 에러메세지를 받아와서 렌더링 시스템에 출력 함
const errorMessageAppend = (message: string) => {
    eventBus.$emit('tokenError', message);
    throw new Error(message);
};

```

```
// 렌더링 시스템의 input 영역에 입력한 코드를 출력함
const lineAppend = (indent: string = '') => EventBus.$emit('lineAppend', indent);

// 렌더링 시스템의 output 영역에 print의 결과를 출력함
const outputAppend = (output: string = '') => EventBus.$emit('outputAppend', output);
```

Parser 호출 정리 : checkState → statement → expression → term → pow → factor

E. Interpreter.vue (Rendering System. Parser를 호출하는 역할)

```
@Component
export default class Interpreter extends Vue {
  private codeList: string[] = []; // input area에 출력될 코드
  private code: string = ''; // 한 line의 코드
  private errorText: string = ''; // error에 대한 내용
  private outputList: string[] = []; // output area에 출력될 코드
  @Watch('codeList') // codeList의 변화를 지켜봄
  private onCodeListChange() {
    const { stackContainer } = any = this.$refs;
    this.$nextTick(() => { // 렌더링 후 스크롤을 맨 밑으로 내림
      stackContainer.scrollTo(0, stackContainer.scrollHeight);
    });
  }
  @Watch('outputList') // outputList의 변화를 지켜봄
  private onOutputListChange() {
    const { outputContainer } = any = this.$refs;
    this.$nextTick(() => { // 렌더링 후 스크롤을 맨 밑으로 내림
      outputContainer.scrollTo(0, outputContainer.scrollHeight);
    });
  }
  private created() { // 태그들이 만들어지는 시점에 실행되는 코드
    const { codeList, outputList } = this;
    // Parser에서 outputAppend를 호출할 경우
    EventBus.$on('outputAppend', (output: string) => {
      outputList.push(output); // output area에 렌더링
    });
    // Parser에서 lineAppend를 호출할 경우
    EventBus.$on('lineAppend', (indent: string) => {
      codeList.push(`${indent}${this.code}`); // input area에 렌더링 후
      this.code = ''; // 입력중인 코드 초기화
      this.errorText = ''; // 에러 텍스트 초기화
    });
    EventBus.$on('tokenError', (message: string) => {
      this.errorText = message; // 에러 처리
    });
  }
  private parsing() { // Parser를 호출하는 코드
    codeContainer.setCode(this.code); // codeContainer에 현재 입력한 코드를 저장 후
    checkState(); // checkState를 통해서 parser를 호출
  }
}
```

F. Helpers.ts

```
import Vue from 'vue';
// Event를 주고받는 Bus를 선언
export const EventBus = new Vue();
// 문자열이 정상적인 숫자형태인지 체크
export const isNumChar = (temp: string): boolean => !isNaN(+temp);
// 문자열이 변수형태인지 정규식으로 체크
export const isVar = (temp: string): boolean => /^[w\$_]+$/i.test(temp);
```