

개요

기본적으로 HW01의 코드를 기반으로 작성하였습니다. 크게 달라진 점은 없어서 기존 HW01 보고서를 HW02 보고서 아래에 첨부하였습니다.

Do - While Statement

문법이 틀린 경우, 표준 출력에 어떠한 출력도 없이 `Syntax Error!!` 만을 출력해야 합니다. 반면에, 무한 반복문이 있으면 `print` 에 의한 출력을 즉시 표준 출력에 flush하여 보여주어야 합니다. 다음과 같은 Language 두 개를 고려해보겠습니다.

```
do { print 1 ; } while ( == 1 1 ) ; // 1이 무한히 출력되는 코드
do { print 1 ; } while ( == 1 1 ) ; print x ; // Syntax Error!!만 출력되는 코드
```

HW01처럼 파싱과 동시에 코드를 실행하면, `print x` 까지 Parser가 넘어가지 못하고 `1` 을 무한히 출력하게 됩니다. (만약, output buffer를 표준 출력이 아니라 따로 둔다고 하더라도, `print x` 를 파싱하지 못하는 것은 같습니다.) 따라서, Parser의 첫 번째 실행에서는 문법이 맞는지만을 확인하고, 그 다음에 실제 코드를 실행해야 합니다.

Introduce `check_grammar_only` in parser

- parser 클래스의 field로 제공됩니다.
- `false` 이면 `print` 문을 실행하지 않으며, `do - while` 구문도 파싱만 하고 넘어갑니다.

Implement(Execute) `do - while` statement

non-terminal `<statement>` 에서 `do` 를 token으로 받으면, 남은 `{ <statement> } while (<bexpr>) ;` 을 순서대로 입력 받고 실행합니다. 그런 후, `<bexpr>` 의 값이 `false` 이면 실행을 중지하고, 그렇지 않으면 `do` token의 다음 토큰(`{ <statement> }`)으로 돌아가서 다시 실행합니다. 이런 식으로 단순히 Parser의 버퍼를 앞으로 돌리는 방식으로 반복문 구현을 하였습니다.

Some Corner Case

```
int int ; int = 1 ; print int ; // 실행 가능
do { print 1 ; } while ( == 1 1 ) ; // 실행 가능

do { print 1 ; } while ( == 1 1 ) ; print x ; // 실행 불가
int a ; print a ; int a ; // 실행 불가
```

- 별다른 제약 조건이 없으므로 `int` 는 변수명으로 사용할 수 있습니다. 따라서, token(lexeme)을 파싱할 시에 `int` 리터럴이면 무조건 `<type>` 이라고 파싱해서는 안 됩니다.
- `do - while` 구문은 위에서 설명한 예와 같습니다.
- `<statement>` 가 온 뒤부터는 `<declaration>` 이 올 수 없습니다.

Program execution result

- Corner case

[illegible]

- Provided example case

```

/tmp/tmp.0X1EDJV46b/cmake-build-debug/hw02
>> int variable ; variable = 365 ;
int variable ; variable = 365 ;

>> ab = 12 ;
ab = 12 ;
Syntax Error!!
>> float ab ;
float ab ;
Syntax Error!!
>> int k = 2 ;
int k = 2 ;
Syntax Error!!
>> int k ; int j ; k = 3 ; j = 20 ; print k + j
int k ; int j ; k = 3 ; j = 20 ; print k + j
Syntax Error!!
>> int k ; int j ; k = 3 ; j = 20 ; int a ; print k + j ;
int k ; int j ; k = 3 ; j = 20 ; int a ; print k + j ;
Syntax Error!!
>> int k ; int j ; k = 3 ; j = 20 ; print k + j ;
int k ; int j ; k = 3 ; j = 20 ; print k + j ;
23
>> int k ; print k + 100 * 3 / 2 - 1 ;
int k ; print k + 100 * 3 / 2 - 1 ;
149
>> int x ; x = 10 + 5 - ( 2 + 3 - 5 * 10 ) ; print x ;
int x ; x = 10 + 5 - ( 2 + 3 - 5 * 10 ) ; print x ;
15
>> int x ; x = 10 + 5 * 2 ; print x ; print > 10 x ;
int x ; x = 10 + 5 * 2 ; print x ; print > 10 x ;
30 FALSE
>> int k ; int j ; k = 3 ; j = 20 ; do { print k + ( j - 1 ) * 10 ; k = k - 1 ; } while ( > k + 10 10 ) ; pr
int k ; int j ; k = 3 ; j = 20 ; do { print k + ( j - 1 ) * 10 ; k = k - 1 ; } while ( > k + 10 10 ) ; print == k 0 ;
220 210 200 TRUE
>> int i ; int j ; i = 0 ; j = 0 ; do { j = j + i ; i = i + 1 ; } while ( < i * ( 5 - 4 ) 5 ) ; print i ; pr
int i ; int j ; i = 0 ; j = 0 ; do { j = j + i ; i = i + 1 ; } while ( < i * ( 5 - 4 ) 5 ) ; print i ; print j ;
5 10
>> terminate
terminate

Process finished with exit code 0

```

개요

하나의 state로 변형되는 자명한 경우를 제외하고, 여러 경우 중 하나로 상태가 바뀌는 경우만 잘 처리해주었습니다. 각 statement, expr의 경우만 자세하게 설명하며, <dec>는 문자를 하나씩 입력 받는 대신에 token 하나를 통째로 입력 받습니다.

다른 언어로 구현된 소스코드보다는 `main.cpp`를 보는 것을 권장합니다.

<statement> 처리

첫번째 토큰을 입력 받은 후, 토큰의 종류가 변수인지, print인지 확인하는 것만으로 어떤 상태로 전이하는지 알 수 있습니다. 따라서, 이 경우는 아래와 같이 처리하였습니다.

```
auto curr = _input.get_token();
if (curr.type == VARIABLE) {
    ...
    return {true, ""};
} else if (curr.type == PRINT) {
    ...
} else {
    return {false, ""};
}
```

<expr> 처리

이 경우는 꽤나 복잡합니다. <bexpr>이나 <aexpr>이나 둘다 첫번째 토큰이 결과적으로 수^{Decimal}를 반환하는 것은 같습니다. 그렇다고 토큰을 하나 더 읽어 연산자만으로 <aexpr>인지 구분하는 것도 구현이 깔끔하지 못합니다. 왜냐하면 <aexpr> → <term> → <factor> → <number>로 전이하는 경우, 연산자 없이도 <aexpr>로 전이할 수 있으며, 다음 연산자가 +, -, *, /로 올 수 있는 연산자가 재귀적으로 정의되기 때문에 "<expr>" 구현에 하위(자손) 상태의 연산자까지 신경 써주어야 합니다. 따라서, 정석적인 방법으로 각각 <bexpr>, <aexpr>로 파싱을 시도합니다. 이를 위해서는 Token Recognizer^{Lexical Analysis}에 롤백^{rollback} 기능을 추가했습니다. "유효하지 않은 방법으로 토큰을 읽어 들였다면", 읽어들인 토큰을 다시 버퍼에 반납하는 방법으로 파싱이 이루어집니다.

최종 코드는 다음과 같은 방식으로 구현하였습니다.

```

pair<bool_int, result> expr() {
    _input.create_restore_point(); // create restore point

    auto res = bexpr();
    if (!res.second.first) {
        _input.restore();
        auto res = aexpr();
        // do rollback token(to restore point) used when tried to parse as bexpr, and then
        try to parse as aexpr
        // if not still, return unknown
    } else {
        // parse as bexpr (oneshot!), and remove restore point
    }
}

```

<program> 출력 처리

한편, print 문은 <statement> 내부에서만 만들어집니다. 하지만, 파싱하는 도중에 print 문을 실행하면 syntax error 임에도 불구하고 출력되는 상황이 발생할 수 있습니다. 따라서, <program>에서는 <statement>의 출력 값을 저장해놓고, EOF를 읽었을 때 한번에 출력하는 방식으로 동작합니다.

그외 주요 부분

Parse a token

```

token_info get_token() {
    if (eof()) return {EOF_, ""};
    string token = get_token_string();
    token_info ret = {UNKNOWN, token};
    if (token == "==" || token == "!=" || token == "<" || token == ">" || token == "<=" ||
    token == ">=") ret.type = OPERATOR_COMP;
    else if (token == "=") ret.type = ASSIGN;
    else if (token == ";") ret.type = SEMICOLON;
    else if (is_numeric(token)) ret.type = DECIMAL;
    else if (token == "print") ret.type = PRINT;
    else if (token == "+" || token == "-") ret.type = OPERATOR_PLUS_MINUS;
    else if (token == "*" || token == "/") ret.type = OPERATOR_MUL_DIV;
    else if (token == "x" || token == "y" || token == "z") ret.type = VARIABLE;
    return ret;
}

```

직접 if - else if 문을 사용하여 처리합니다. 간단하니 설명은 생략합니다. 앞뒤에 있는 공백을 모두 읽어들이고 후에 token 입력을 하나 받습니다.

<expr>의 반환값 처리

<expr>은 <bexpr> 또는 <aexpr>로 변경됩니다. 따라서, 반환 값은 수^{Decimal}이거나 부울^{boolean} 형식입니다. 적당히 C++의 union과 struct를 사용해서 현재 저장하고 있는 값이 bool인지, decimal인지, 그렇다면 값은 무엇인지를 저장하도록 했습니다.

```

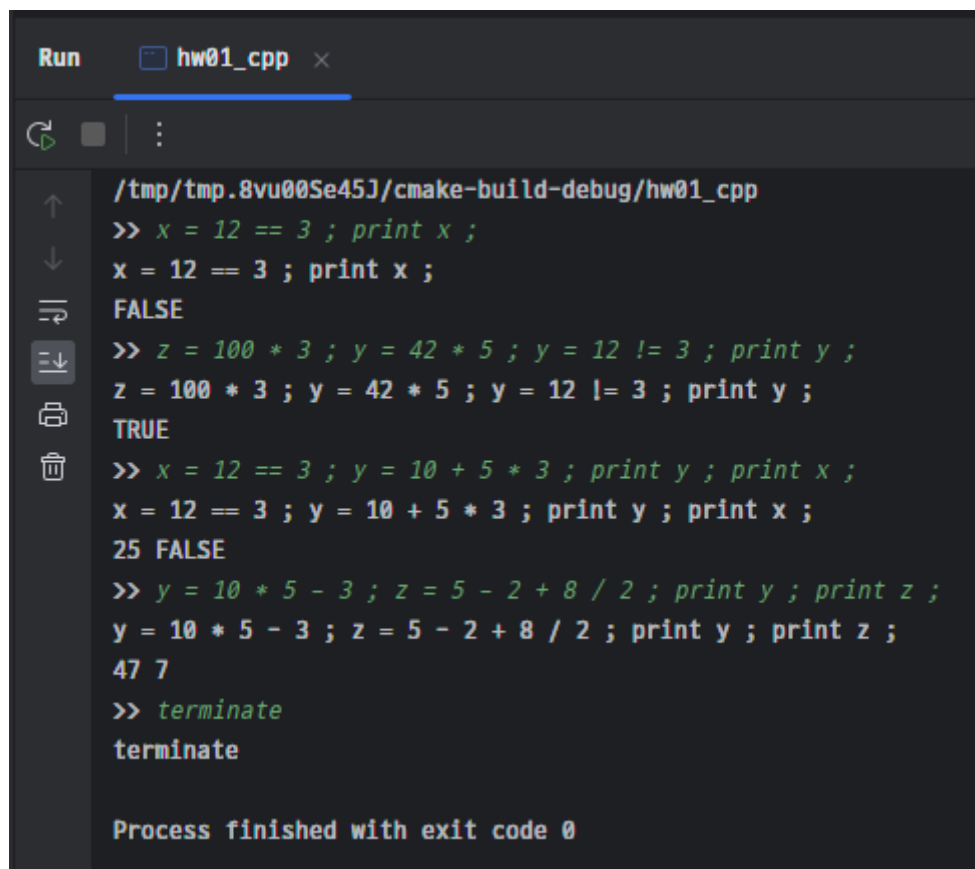
struct bool_int {
    int is_bool;
    union {
        int i;
        bool b;
    } data;
};

string to_string(const bool_int &a) {
    return a.is_bool ? (a.data.b ? "TRUE" : "FALSE") : to_string(a.data.i);
}

```

실행 사진

Valid syntax



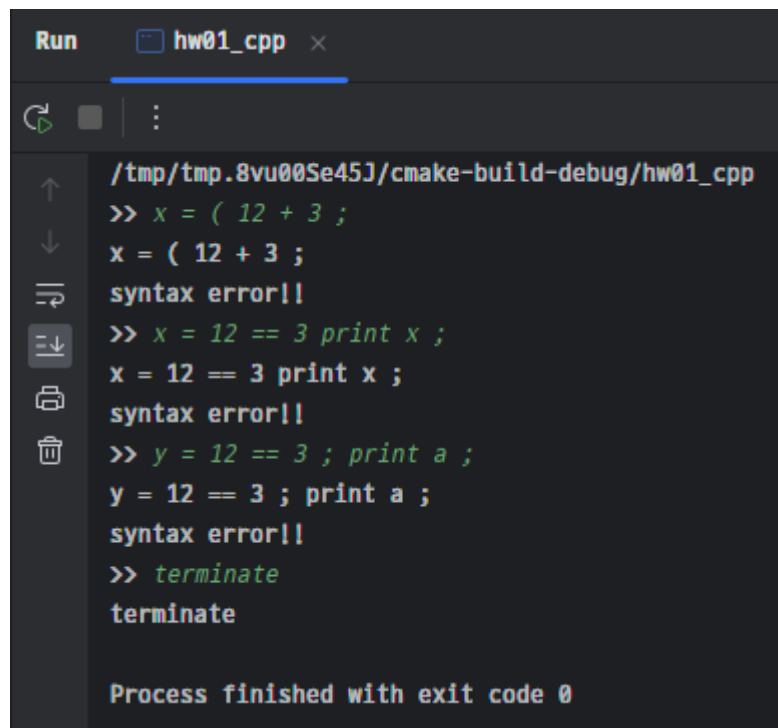
```

Run    hw01_cpp x
/tmp/tmp.8vu00Se45J/cmake-build-debug/hw01_cpp
>> x = 12 == 3 ; print x ;
x = 12 == 3 ; print x ;
FALSE
>> z = 100 * 3 ; y = 42 * 5 ; y = 12 != 3 ; print y ;
z = 100 * 3 ; y = 42 * 5 ; y = 12 != 3 ; print y ;
TRUE
>> x = 12 == 3 ; y = 10 + 5 * 3 ; print y ; print x ;
x = 12 == 3 ; y = 10 + 5 * 3 ; print y ; print x ;
25 FALSE
>> y = 10 * 5 - 3 ; z = 5 - 2 + 8 / 2 ; print y ; print z ;
y = 10 * 5 - 3 ; z = 5 - 2 + 8 / 2 ; print y ; print z ;
47 7
>> terminate
terminate

Process finished with exit code 0

```

Invalid syntax



The screenshot shows a terminal window titled "Run" with a tab for "hw01_cpp". The terminal output shows the execution of a C++ program with several syntax errors. The errors are highlighted in red in the original image. The program attempts to perform arithmetic and assignment operations, but the syntax is incorrect. The errors are as follows:

```
/tmp/tmp.8vu00Se45J/cmake-build-debug/hw01_cpp
>> x = ( 12 + 3 ;
x = ( 12 + 3 ;
syntax error!!
>> x = 12 == 3 print x ;
x = 12 == 3 print x ;
syntax error!!
>> y = 12 == 3 ; print a ;
y = 12 == 3 ; print a ;
syntax error!!
>> terminate
terminate

Process finished with exit code 0
```