MiniC Scanner

2020112082 오현석

형식언어

송수환 교수님

2024.05.19

MiniC의 코드를 토큰으로 나누어 parser에게 전달해주는 역할을 가진 Scanner를 부분 구현하고 그에 대한 테스트 파일에 대한 결과 분석을 진행하겠다. 토큰을 정의하는 'Token' 클래스, 토큰을 생성하는 'Scanner'클래스, 그리고 다양한 토큰 타입을 정의하는 'TokenType' 열겨형으로 이루어져 있다.

1. 코드 설명

Token.java

```
private Token (TokenType t, String v) {
    type = t;
    value = v;
    if (t.compareTo(TokenType.Eof) < 0) {
        int ti = t.ordinal();
        reserved[ti] = v;
        token[ti] = this;
    }
}
```

Token 클래스는 여러가지 토큰의 타입과 값을 정의한다. 클래스 내에 여러가지 토큰에 대한 타입과 값을 저장한 객체를 생성하여 Scanner에서 사용할 수 있게 한다.

```
public static final Token whileTok = new Token(TokenType.While, "while");
public static final Token charTok = new Token(TokenType.Char, "char");
public static final Token doubleTok = new Token(TokenType.Double,
"double");
public static final Token forTok = new Token(TokenType.For, "for");
public static final Token doTok = new Token(TokenType.Do, "do");
public static final Token gotoTok = new Token(TokenType.Goto, "goto");
public static final Token switchTok = new Token(TokenType.Switch,
"switch");
public static final Token caseTok = new Token(TokenType.Case, "case");
public static final Token breakTok = new Token(TokenType.Break, "break");
public static final Token defaultTok = new Token(TokenType.Default,
"default");
```

따라서, 키워드 char, double, for, do, goto, switch, case, break, default를 추가하기 위해서는 해당 토큰에 대한 객체를 생성할 필요가 있었다.

```
public static Token mkStringLiteral (String name) {
    return new Token(TokenType.StringLiteral, name);
}

public static Token mkDoubleLiteral (String name) {
    return new Token(TokenType.DoubleLiteral, name);
}
```

또한, string과 character는 추가로 인식해줘야 하는 리터럴이기 때문에 각각 mkStringLiteral과 mkDoubleLiteral 메서드를 정의하여 Scanner에서 특정 조건에서 string이나 character임을 인식하게 된다면 이 메서드를 호출하여 'StringLiteral'과 'DoubleLiteral' 로 인식하게 끔 하였다.

Scanner.java

```
private char nextChar() { // Return next char
    if (ch == eofCh)
        error("Attempt to read past end of file");
    col++;
    if (col >= line.length()) {
        try {
            line = input.readLine( );
        } catch (IOException e) {
            System.err.println(e);
            System.exit(1);
        } // try
        if (line == null) // at end of file
            line = "" + eofCh;
        else {
            // System.out.println(lineno + ":\t" + line);
            lineno++;
            line += eolnCh;
        } // if line
        col = 0;
    } // if col
    return line.charAt(col);
}
```

Scanner클래스에서 다음 문자를 읽어오는 메서드이다. 파일에서 한 줄씩 읽어오는 방식으로 읽기를 실패했다면 파일의 끝으로 간주하며 읽어온 문장의 가장 첫번째 문자를 반환하여 필요할 때 다음 문자를 제공한다.

```
else if (isDigit(ch) || ch == '.') { // int or double literal
    String number = "";
    boolean isDouble = false;

    if (ch == '.') {
        isDouble = true;
        number += ch;
        ch = nextChar();
    }

    number += concat(digits);

    if (ch == '.') {
        isDouble = true;
        number += ch;
        ch = nextChar();
        if (isDigit(ch)) {
            number += concat(digits);
        }
    }

    if (isDouble) return Token.mkDoubleLiteral(number);
    else return Token.mkIntLiteral(number);
```

다음은 next 메서드 중 정수 혹은 실수를 읽는 부분이다. next 메서드는 nextChar 메서드로 받아온 문자의 타입을 판별하여 해당하는 토큰 타입과 값을 반환하는 메서드이다. isDouble이란 boolean형 변수를 이용하여 실수와 정수를 구분해낸다. 일련의 과정 중 해당 숫자가 정수이면 IntLiteral을, 실수라면 DoubleLiteral을 반환한다. 이때 double이 .123 이나 123. 과 같은 숏폼도 인식해야한다. 첫 번째 문자가 '.' 이라면 이는 실수를 의미함으로 isDouble을 true로 바꾸고 그렇지 않다면 number에 값(숫자)를 붙인다. 그 다음 문자가 '.' 라면 이 경우도 실수를 의미함으로 isDouble을 true로 바꾸고 다음 문자를 number에 붙인다. 이 과정을 지나고 완성된 number 문자열을 isDouble의 값에 따라 true면 mkDoubleLiteral을, 그렇지 않다면 mkIntLiteral을 호출하여 해당하는 토큰 타입을 반환할 수 있게 한다.

```
case '\'':  // char literal
    char ch1 = nextChar();
    nextChar(); // get '
    ch = nextChar();
    return Token.mkCharLiteral("" + ch1);

case eofCh:
    return Token.eofTok;

case '\"': // string literal
    ch = nextChar();
    String str = "";
    while (ch != '\"') {
        str += ch;
        ch = nextChar();
    }
    ch = nextChar();
    return Token.mkStringLiteral(str);
```

다음은 next 메서드 중 character과 string을 인식하는 부분이다. 먼저 character는 single quote(') 로 시작할 때 인식된다. single quote로 시작한다면, 다음 문자를 불러올 때 그 값이 곧 character 형 문자임을 의미한다. 그 값을 앞서 정의한 mkCharLiteral의 인자로 해당하는 토큰 타입과 값을 반환한다.

string은 double quote(")로 시작할 때 인식된다. Double quote로 시작한다면, 다음 문자는 문자열이다. 다만 character와 다른 점은 문자로 이루어진 문자열이기 때문에 다음 토큰이 double quote가 나올 때까지 str에 붙여서 하나의 문자열을 완성시켜야 한다. 다음 문자가 double quote가 나오면 그 문자열은 끝이 났음을 의미한다. 그렇게 완성된 str을 앞서 정의한 mkStringLiteral의 인자로 해당하는 토큰 타입과 값을 반환한다.

```
case ':':
    ch = nextChar();
    return Token.colonTok;
```

next 메소드의 콜론을 인식하는 부분이다. 이는 switch, goto, default 문에 필요하여 정의하였다.

2. 결과화면

반복되는 부분은 추가하지 않고 각 파일 마다 Scanner의 기능을 보여줄 수 있는 부분만을 선택하여 작성하였다.

comment.mc

```
Begin scanning... programs/comment.mc

void
Identifier  main
(
)
{
int
Identifier  i
;
Identifier  i
=
IntLiteral  100
;
Identifier  write
(
Identifier  i
)
;
}
```

간단한 main함수에서 int 형 변수를 선언 및 값을 할당하고 write함수를 호출하는 코드이다. 함수, 변수이름은 타입이 identifier로, 정수는 Intliteral로 설정되고 다른 토큰들은 적절하게 분리되었음을 볼 수 있다.

ext.mc

```
int
Identifier  x
;
void
Identifier  main
(
)
{
Identifier  func
(
)
;
Identifier  write
(
Identifier  x
)
;
}
void
Identifier  func
(
)
{
Identifier  x
=
IntLiteral  100
;
}
```

새로운 함수, func이 정의된 코드가 추가된 파일이다. 함수의 반환형과 이름이 잘 분리되었고 함수 이름은 identifier로 정의되었음을 확인할 수 있다.

factorial.mc

```
void
Identifier  main
(
)
{
int
Identifier  n
,
Identifier  f
;
Identifier  read
(
Identifier  n
)
;
Identifier  write
(
Identifier  n
)
;
Identifier  f
=
Identifier  factorial
(
Identifier  n
)
;
Identifier  write
(
Identifier  f
)
;
}
int
Identifier  factorial
(
int
Identifier  n
)
{
if
(
Identifier  n
==
IntLiteral  1
)
return
IntLiteral  1
;
else
return
Identifier  n
*
Identifier  factorial
(
Identifier  n
-
IntLiteral  1
)
;
}
```

팩토리얼을 재귀적인 방식으로 구하는 예시 파일이다. 재귀함수를 반환 시 조건문으로 if문이 사용되었다. if문의 조건 부분에 비교 연산자 '=='가 n과 1 사이에서 토큰으로 분리가 됨을 볼 수 있다. if문이 원하는대로 분리되었다.

mod.mc

```
Identifier  x
%=
IntLiteral  10
;
```

'%='과 같은 복합 할당 연산자도 identifier와 literal 사이에서 잘 분리된 모습이다.

pal.mc

```
if
(
Identifier  org
<
IntLiteral  0
)
Identifier  org
=
(
-
IntLiteral  1
)
*
Identifier  org
;
Identifier  i
=
Identifier  org
;
Identifier  rev
=
IntLiteral  0
;
while
(
Identifier  i
!=
IntLiteral  0
)
{
Identifier  j
=
Identifier  i
%
IntLiteral  10
;
Identifier  rev
=
Identifier  rev
*
IntLiteral  10
+
Identifier  j
;
Identifier  i
/=
IntLiteral  10
;
}
```

여기서는 while문이 사용되었다. while문 조건 부분의 != 과 while문 내의 할당 연산자 /=를 비롯한 *, +, =, % 등의 연산자들이 잘 분리되어 하나의 토큰 값을 가지는 모습을 확인할 수 있다.

perfect.mc

```
const
int
Identifier  max
=
IntLiteral  500
;
```

상수 값을 나타내는 const도 하나의 토큰으로 잘 분리되었다.

extended_test.mc (추가한 기능에 대한 테스트)

extended_test.mc는 추가된 심볼들과 추가 인식되어야 할 리터럴(char, string, double)에 대해 테스트하기 위한 테스트 파일이다.

```
int main() {

    char c = 'A';
    double d = 3.14;
    double e = .123;
    double f = 123.;
    string g = "abcdefg";

    for (int i = 0; i < 5; i++) {
        i = i + 1;
    }

    int j = 0;
    do {
        j++;
    } while (j < 3);

    goto skip;
    printf("This will be skipped.\n");
    skip:
    printf("This will be printed.\n");

    int x = 2;
    switch (x) {
        case 1:
            printf("Case 1\n");
            break;
        case 2:
            printf("Case 2\n");
            break;
        default:
            printf("Default case\n");
    }

    return 0;
}
```

결과화면)

```
int
Identifier  main
(
)
{
char
Identifier  c
=
CharLiteral A
;
double
Identifier  d
=
DoubleLiteral   3.14
;
double
Identifier  e
=
DoubleLiteral   .123
;
double
Identifier  f
=
DoubleLiteral   123.
;
```

double, char

single quote로 시작된 값들은 CharLiteral로, 숏폼(.123, 123.)을 포함한 실수 값들은 DoubleLiteral의 토큰 타입을 가지며 다른 연산자, identifier와 분리됨을 확인할 수 있다.

```
DoubleLiteral   123.
;
Identifier  string
Identifier  g
=
StringLiteral   abcdefg
;
for
(
int
Identifier  i
=
IntLiteral  0
;
Identifier  i
<
IntLiteral  5
;
Identifier  i
++
)
{
Identifier  i
=
Identifier  i
+
IntLiteral  1
;
}
```

string, for문

double quote로 시작된 토큰은 문자열로 StringLiteral로 분류되었다. for문 또한 초기화, 조건, 증감 연산자 모두 세미콜론과 연산자 그리고 (), {} 으로 분리됨을 확인할 수 있다.

```
int
Identifier  j
=
IntLiteral  0
;
do
{
Identifier  j
++
;
}
while
(
Identifier  j
<
IntLiteral  3
)
;
```

do-while문

do 키워드를 인식하고 분리한다.

```
goto
Identifier  skip
;
Identifier  printf
(
StringLiteral   This will be skipped.\n
)
;
Identifier  skip
:
Identifier  printf
(
StringLiteral   This will be printed.\n
)
```

goto문

goto 키워드를 인식하고 토큰으로 분리한다. 여기서 skip 뒤의 콜론(:) 도 하나의 토큰으로 인식되어 분리됨을 확인할 수 있다.

```
switch
(
Identifier  x
)
{
case
IntLiteral  1
:
Identifier  printf
(
StringLiteral   Case 1\n
)
;
break
;
case
IntLiteral  2
:
Identifier  printf
(
StringLiteral   Case 2\n
)
;
break
;
default
:
Identifier  printf
(
StringLiteral   Default case\n
```

switch, default

switch와 switch내의 case 및 default도 하나의 키워드로 인식하고 토큰으로 분리된다. 마찬가지로 case와 default의 경우 콜론(:)을 포함하고 있기 때문에 콜론도 하나의 토큰으로 분리됨을 확인할 수 있다.

3. 소감

프로그램의 기본 단위인 토큰의 구조를 이해하고, 각 토큰이 가져야 할 값들에 대해 정의하는 과정들에 있어서 전체 코드를 파악하고 각각의 메소드의 역할을 이해하는 과정들은 필수적이었다. 이렇게 좋은 객체지향적 코드를 보고 어디가 어디로 이어지는지에 대해 파악하는 과정들이 좋은 개발자에 한 발자국 다가간 느낌이다. 또한 scanner 클래스에서 특정 조건에서 특정 토큰을 인식하게 하기 위해서는 적절한 알고리즘을 작성해야 했다. 이는 다양한 MiniC 토큰 구조에 대해 익숙해지는 기회가 되었다. 개인적으로 이런 과제가 더 필요하다고 생각한다. 간단한 코드들은 AI가 생성해주는 시대에서 오픈소스를 보고 어떤 부분을 어떻게 사용하고 수정할지 결정하는 것은 개발자이기 때문에 결국 많은 코드를 보고 작동법을 익히는 이 일련의 과정들이 본인에게는 더 필요할 것 같다.