JSCC 준비

HandyPost는 한 도영(HDNua)이 작성하는 포스트 문서입니다.

1. 개요

jscc를 개발하기 위한 준비를 한다. jscc로 코드를 번역하거나 실행할 때 기존에 존재하는 어셈블러 프로그램(nasm)을 이용하지 않는다.

2. 사용할 도구

이 절에서 다루는 도구에 대한 사용법을 보려면 필자의 블로그[1]를 참조하라.

2.1) Brackets

이 프로젝트를 포함한 이후의 문서에서는 개발 도구는 모두 Brackets만 사용한다. 이전 문서를 읽으면서 Brackets의 사용법을 익혔을 것이므로 여기에서 별도로 설명하지 않는다.

2.2) Node.js

부대 내에서 컴파일러를 개발할 때는 IE의 ActiveX를 이용하여 Handy HTML Maker라는 html 문서를 작성하여 프로그램을 만들고, 이를 이용하여 파일에 접근했다. 하지만 이미 익히 알려진 바와 같이 ActiveX는 사회악이고 축출될 위기에 처해있다. 현대의 브라우저에 내장된 JavaScript는 파일 시스템에 접근할 방법이 없다. 따라서 브라우저에서 바로 파일에 데이터를 기록하는 프로그램은 생성할 수 없다.

Node.js는 이를 위해 사용한다. Windows와 Mac OS X 환경이라면 자동 설치 도구가 있고, Linux를 이용하는 사람도 바이너리가 제공된다. 어차피 우리는 Node.js의 깊은 부분을 자세하게 다루는 것이 아니라, JavaScript를 이용하여 컴파일러를 개발하기 위한 JavaScript 실행기로써만 Node.js를 다루는 것이기 때문에 이에 대한 사전 지식은 전혀 필요하지 않다. 그저 버튼 몇 번 눌러서 설치만 하면 된다.

2.3) nw.js

Node.js를 이용하여 GUI 프로그램을 만들기 위한 확장 프로그램이다. 이전에는 Node-webkit이라는 프로젝트였지만 이름이 바뀌었다. 이 파일은 설치 파일은 없고 바이너리만 제공되는데 그대로 사용할 수 있으므로 크게 걱정하지 말자.

이 정도면 jscc 개발을 위한 첫 발을 성공적으로 내딛었다고 할 수 있다.

3. 왜 nasm을 이용하지 않나요?

이미 좋은 어셈블러와 링커가 있는데도 불구하고 어셈블러와 링커를 손수 제작한다는 점이 못마땅할 수 있다. 이에 대해 지금 이야기를 풀어야 앞으로 하는 모든 코딩을 납득할 수 있을 것이므로 이 자리에서 이에 대해 설명하는 것이 좋겠다.

3.1) JSCC는 순수하게 논리만으로 컴파일러를 개발하는 것이 가능함을 보이는 프로젝트입니다.

이 프로젝트는 컴파일러를 개발하고 싶은 초·중급 프로그래머를 위한 것이다. 일단 우리가 실행해서 사용하고 있으니 제대로 만들 수 있는 건 맞지만, 정말 그것이 가능한가를 증명함으로써 가슴 한 구석에 품고 있던 막연한 궁금증을 푸는 실마리가 된다. 또한 다른 도구를 이용하지 않고 컴파일러를 개발해냈다는 나름의 자부심이 생길 것이다.

3.2) 우리가 무엇을 하는지를 정확히 이해하게 됩니다.

단순히 컴파일러를 어떻게 만들 수 있는지를 이해할 수 있는 것만이 아니다. 인간의 코드가 기계어로 번역되는 과정을 이해하면, 다른 고급 언어를 사용하여 코드를 작성할 때 이것이 어떤 이유에서 그렇게 구현되었는가를 보다 명확하고 논리적으로 이해할 수 있게 된다.

3.3) 프로그램 디버깅, 리버스 엔지니어링에 큰 도움이 됩니다.

역공학에 능하려면 우선 정공학에 능해야 한다는 건 자명하므로 자세하게 설명하지 않아도 될 것이다. 리버싱은 이미 만들어진 프로그램을 다른 도구를 이용하여 분석하는 공학을 이야기하는데, 올리디버거와 같은 도구를 사용한다면 프로그램을 실행하자마자 화면에 수십만 개의 어셈블리 코드를 만나게 된

1) http://hdnua.tistory.com/17

다. 우리가 컴파일러를 개발할 때는 어셈블리 언어에 싫어도 익숙해질 수밖에 없고, 요즘 웬만한 프로그램은 고급 언어로 작성되었으며 컴파일러는 나름의 방법으로 고급 언어를 저급 언어로 번역한다. 따라서 컴파일러가 고급 언어 코드를 번역하는 패턴을 파악한다면 완성된 프로그램을 열어 어셈블리 코드를 분석할 때 어떤 부분이 지역 변수의 선언이며 어떤 부분이 루프를 형성하는지를 이해하는 데 크게 도움을 준다.

이 정도면 기존의 프로그램을 이용하지 않는 것에 대해 어느 정도 그럴 만한 이유가 있다고 받아들일 수 있으리라 믿는다. 그래도 이러한 내용이 필요하지 않다고 생각한다면 후에 연재할 컴파일러 개발에 관한 문서만 참조하면 될 것이다.

4. Handy Assembly: HASM 소개

4.1) 개요

이전에 필자가 C와 어셈블리 언어의 중간 언어인 Handy CIL을 소개한 바 있다. 여기서도 같은 일이 일어난다. NASM 어셈블리 언어는 완전한 어셈블리 언어지만, 사실 우리는 NASM의 명령어를 몇 개 알고 있을 뿐 NASM 어셈블리 언어 전체(specification)를 알고 있다고 말할 수는 없다. 강조하지만 우리가 공부한 건 NASM의 정말 일부분일 뿐이다. 우리의 목표는 C 컴파일러고, C 컴파일러는 C 코드를 기계어로 번역하기 위해 어셈블리 언어로 변환해야 한다. 그런데 우리가 온전히 알지 못하는 언어로 변환을 시도한다는 건 어딘가 마음 한 구석이 찜찜해지는 것이다. 내부 구현을 모른 채로 STL의 스택과 같은 자료구조를 꺼내서 쓴 다음, 버그가 발생하면 STL의 스택이 버그인지, 자신의 코드가 버그인지 판단할 수 없는 상황이 올 수 있다는 것이다(물론 이 경우 99.9%는 자기 잘못이지만).

필자가 HASM이라는 언어를 새롭게 고안했던 건 인트라넷 컴퓨터에 어셈블러가 없어서 이것도 스스로 개발해야 했던 탓이지만, 여기서는 자신의 코드에 신뢰감을 더하기 위한 것이라고 말하겠다. 우리는 HASM이라는 언어에 필요한 명령을, 필요할 때 추가하고 구현할 것이다. 이 과정은 꽤 재밌다. 지금 당장 무엇을 하는지 이해하지 못하더라도 프로젝트를 진행하다보면 자연히 알게 되는 것이니 고민하지 않아도 좋다.

4.2) HASM 뼈대 파일

HASM은 이름이 Handy Assembly인 만큼, 기본적으로 어셈블리 언어에 뿌리를 두고 있다. 그래서 거의 모든 구조가 기존의 어셈블리 언어를 잘 따르고 있다.

예제를 보이기 전에 파일 확장자 세 개를 설명하겠다. hda 파일은 Handy Assembly의 약자로, HASM 어셈블리 언어로 작성된 소스 코드 파일이다. hdo 파일은 Handy Object의 약자로, 우리가 개발할 어셈블러 모듈이 hda 어셈블리 소스 코드를 변환하여 생성되는 목적 파일이다. hdx는 Handy Executable의 약자로, 마찬가지로 우리가 개발할 링커 모듈이 어셈블러를 거쳐 생성된 목적 파일을 묶어 프로그램으로 만든 것이다. 다시 말하면 hdx 파일은 프로그램, 즉 실행 가능한 목적 파일이다.

그러면 바로 HASM 어셈블리 언어로 작성된 뼈대 파일을 보이겠다.

HelloHASM.hda

```
; 데이터 세그먼트의 시작을 나타냅니다.
; nasm과 다르게 segment 지시어를 기록하지 않습니다.
.data

; 코드 세그먼트의 시작을 나타냅니다.
.code
_main:
push ebp
mov ebp, esp
```

```
; put your code here

mov esp, ebp
pop ebp
ret
```

이 코드를 보고 나선 실망했을 수도 있다. NASM과 다른 것이 크게 없기 때문이다. 하지만 우리가 할 일은 이를 nasm과 같은 프로그램에 넘기는 것이 아니라 이 코드를 해석하고 실행하는 실행기를 만드는 일이다. 그래서 “Hello, world!"와 같은 기본 문자열을 출력하는 것도 이 코드에서는 제외했다.

이 코드가 HASM의 전체라면 당연히 고급 프로그램을 작성하는 것이 불가능하다. 이 문서에서는 일단 이렇게 간단한 코드를 해석하는 프로그램을 만든 다음, 필요한 기능이 있을 때마다 실행기에 기능을 추가하는 식으로 프로그램을 만들어나갈 것이다.

5. 프로젝트 준비

여기서는 어셈블리 실행기인 Runner를 개발하는 것을 목표로 한다.

5.1) 메모리

프로그램은 메모리를 사용한다. 따라서 우리가 생성할 실행 파일도 메모리를 사용한다. 메모리는 바이트의 배열이다. 이들을 합치면 우리가 생성할 실행 파일은 바이트의 배열을 사용한다는 결론이 나온다. 즉 우리는 바이트의 배열을 표현하는 객체를 만들고, 이 객체에 접근하여 메모리에서 값을 가져오거나 메모리에 값을 기록해야 한다.

사실 구체적으로 프로그램이 메모리를 어떻게 사용하는가에 대해서는 4장에서 이미 다룬 바 있다. 4장에서 CIL을 배우면서 지역 변수를 만들 때나 함수를 호출하고 원래 주소로 복귀할 때, 각 명령이 어떤 방식으로 메모리에 접근하는지를 그림으로 학습했다(이 부분이 기억나지 않는다면 다시 학습하고 와야 한다). 여기서는 이를 고려해서 메모리를 표현하는 바이트 배열을 실행기의 필드로 추가할 것이다.

```
this.mem = new Array(MAX_MEMORY_SIZE);
```

그럼 이제 본격적으로 실행기 모듈을 작성해보자.

5.2) 로그 스트림

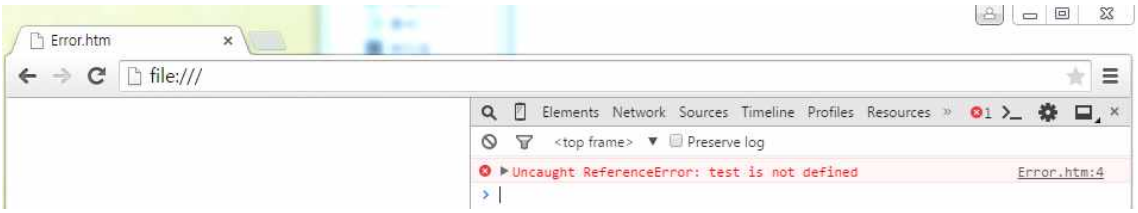
5장의 QuickNASM을 사용하면서도, 6장에서 NASM을 배울 때도 우리는 항상 로그 스트림을 만들었다. 왜냐하면 로그 스트림은 프로그램에서 발생한 예외를 화면을 옮기지 않고 바로 알아볼 수 있도록 하는 아주 편리한 도구이기 때문이었다. 따라서 여기서도 로그 스트림을 사용한다.

로그 스트림을 만드는 방법은 이미 6장을 공부해서 잘 알고 있다. 다만 이전에 만든 main 뼈대 파일은 예외가 발생하면 어디서 예외가 발생했는지를 알아보기 힘들기 때문에 이 문제를 개선하는 것이 좋겠다. 예를 들어 다음의 코드는 정의되지 않은 식별자를 사용했기 때문에 오류가 발생한다.

Error.htm

```
<html>
  <body>
    <script>
      test;
    </script>
  </body>
</html>
```

실행 결과



test가 정의되지 않았다는 예외가 catch되지 않아서 콘솔 창에 로그가 출력된다. 사실 Chrome 브라우저는 F12 버튼을 누르면 개발자 모드로 들어가고 내부적으로 로그 스트림을 지원한다. 다만 이 프로젝트에서는 이것을 통해 로그를 확인하는 것이 불편하다고 생각했기 때문에 별도로 스트림을 만든 것이다. 이렇게 문법 오류 등의 이유로 브라우저 내부에서 발생하는 예외 객체는 일반적으로 Error 형식이 된다.

잠깐 생각해보자. C++에서는 try-catch 구문을 쓸 때 예외의 형식에 따라 루틴을 구분할 수 있었다.

TryCatch1.cpp

```
#include <iostream>

// 일반 예외 형식 Exception을 정의합니다.
class Exception {
    std::string _msg;
public:
    Exception(const std::string &msg) : _msg(msg) {}
    const char *toString() const { return _msg.c_str(); }
};
// null 포인터를 참조하는 경우 반환할 예외 클래스입니다.
class NullPointerException: public Exception {
public: NullPointerException() : Exception("포인터가 null을 참조합니다.") {}
};
// 0으로 나누려고 한 경우 반환할 예외 클래스입니다.
class DivideWithZeroException: public Exception {
public: DivideWithZeroException() : Exception("0으로 나누려고 했습니다.") {}
};

int main(void) {
    try {
        int n = 5, d = 0; // 분자(numerator), 분모(denominator) 변수
        int *ptr = nullptr;

        // 포인터가 분모 변수를 가리키게 합니다.
        ptr = &d; // 이 문장을 주석 처리해보세요!
        if (ptr == nullptr) // 포인터가 null이라면 예외 처리합니다.
            throw NullPointerException();
        std::cin >> (*ptr); // 포인터를 이용하여 사용자로부터 값을 입력받습니다.
```

```
        // 입력받은 수를 이용하여 나눗셈 연산을 시도합니다.
        if (d == 0) // 분모가 0이라면 예외 처리합니다.
            throw DivideWithZeroException();
        // 결과를 출력합니다.
        std::cout << n << " / " << d << " = " << (n / d) << std::endl;
        return 0;
    }
    catch (DivideWithZeroException &ex) {
        std::cout << ex.toString() << std::endl;
        std::cout << "0으로 나누지 마세요." << std::endl;
        return 4;
    }
    catch (NullPointerException &ex) {
        std::cout << ex.toString() << std::endl;
        std::cout << "null 포인터를 참조하지 마세요." << std::endl;
        return 3;
    }
}
```

이렇게 형식에 따라 루틴을 분리할 수 있다면 코드의 가독성이 높아진다. 따라서 이는 아주 바람직한 방법이다. 그런데 저번 문서에서 말했듯 JavaScript의 예외 처리 구문에서는 형식을 기록하지 않는다. 그렇다면 JavaScript에서는 어떻게 예외를 구분해야 할까?

방법은 두 가지가 있다. 다른 객체와 절대로 중복되지 않을 이름을 하나 골라서 필드로 만들고, 이 필드의 값을 적당한 값으로 설정하면 일반적인 형식과 이 필드가 있는 형식을 구분하는 것이 가능하다. 예를 들어 다음과 같이 하면 적당한 예외 처리 모델이 된다.

Exception1.htm <html.body.script>

```
/**
jscc 일반 예외 형식 Exception을 정의합니다.
@param {string} msg
*/
function Exception(msg) {
  // 예외에 대한 설명을 보관하는 문자열입니다.
  this.message = msg;
  // 다른 멤버와 중복되지 않을 필드를 만들고 적당히 설정합니다.
  this.handy = true;
}

// 0으로 나누려고 시도하면 예외를 반환하는 예제입니다.
try {
  var n = 10, d = 0;

  // d = 5; // 주석을 해제해보세요!
  if (d == 0) { // 0으로 나누려고 하면 예외를 발생합니다.
    throw new Exception("Tried to divide number with 0");
```

```
  }
  alert(n / d); // 0으로 나눈 결과를 출력합니다.

  // 일반 예외 형식인 Error 객체를 발생합니다.
  test;

} catch (ex) {
  // handy 필드가 정의되지 않았다면 Exception 객체가 아닙니다.
  if (ex['handy'] == undefined) {
    alert('Catched object is not instance of Exception');
  }
  // 그 외의 경우 Exception 객체로 간주하고 처리합니다.
  else {
    alert(ex.message);
  }
}
```

다른 방법은 instanceof 연산자를 이용하는 것인데 다음과 같다.

Exception2.htm <html.body.script>

```
function Exception(msg) {
  this.message = msg;
}

try {
  var n = 10, d = 0;

  // d = 5; // 주석을 해제해보세요!
  if (d == 0) { // 0으로 나누려고 하면 예외를 발생합니다.
    throw new Exception("Tried to divide number with 0");
  }
  alert(n / d); // 0으로 나눈 결과를 출력합니다.

  // 일반 예외 형식인 Error 객체를 발생합니다.
  test;

} catch (ex) {
  // ex가 Exception의 인스턴스인 경우의 처리입니다.
  if (ex instanceof Exception) {
    alert(ex.message);
  }
  // 그 외의 경우에 대한 처리입니다.
  else {
    alert('Catched object is not instance of Exception');
  }
```

```
}
```

여기에서 알 수 있듯이 불필요한 멤버를 추가하는 것보다 instanceof 연산자를 이용하는 것이 코드도 깔끔해지고 목적도 명확하므로 instanceof 연산자를 사용하는 것이 바람직하다고 할 수 있다.

그럼 이제 예외 처리가 고려된 새로운 뼈대 파일을 보이겠다.

main.htm

```
<html>
  <head>
    <script>
      function main() { }
      function init() {
        var logStream = document.getElementById('HandyLogStream');
        logStream.style.width = 800;
        logStream.style.height = 200;
      }

      /**
      @param {string} msg
      @param {object} data
      */
      function Exception(msg, data) {
        this.description = msg;
        this.data = (data != undefined) ? data : null;
      }
      Exception.prototype.toString = function() {
        var type = 'Exception: ';
        var message = this.description;
        var data = (this.data != undefined) ? this.data.toString() : '';
        return type + message + ' [' + data + ']';
      }

      /**
      로그 스트림에 문자열을 출력합니다.
      @param {string} message
      */
      function log(message) {
        var logStream = document.getElementById('HandyLogStream');
        logStream.value += (message + '\n');
      }
    </script>
  </head>
  <body>
    <textarea id='HandyLogStream'></textarea>
    <script>
```

```
    try {
      init();
      main();
    } catch (ex) {
      if (ex instanceof Exception) {
        log(ex);
      } else {
        // Exception 예외 객체가 아니라면 여기서 처리하지 않습니다.
        throw ex;
      }
    }
  </script>
 </body>
</html>
```

그런데 이렇게 써놓고 보니 파일 하나에 코드가 지나치게 길다. init과 main은 우리가 수정할 함수이니 htm 파일에 있는 것이 편하다고 쳐도, log 함수와 Exception 생성자 함수는 앞으로 수정할 일이 없는 것들이다. 아무래도 파일 단위로 코드를 분리할 수 있다면 좋겠다.

JavaScript 코드를 파일로 분리할 때는 <script> 태그의 속성인 src를 이용한다. 다음은 Exception 생성자와 log 함수의 정의를 바깥으로 뜯어낸 것이다.

main.htm <html.head>

```
<!-- 기존 코드를 가져올 때 src 속성을 이용합니다. -->
<script src="handy.js"></script>
<!-- 필요한 코드를 새롭게 추가합니다. -->
<script>
  function main() {
   log("Hello, world!");
  }
  function init() {
    var logStream = document.getElementById('HandyLogStream');
    logStream.style.width = 800;
    logStream.style.height = 200;
  }
</script>
```

handy.js

```
/**
@param {string} msg
@param {object} data
*/
function Exception(msg, data) {
  this.description = msg;
  this.data = (data != undefined) ? data : null;
```

```
}
Exception.prototype.toString = function() {
  var type = 'Exception: ';
  var message = this.description;
  var data = (this.data != undefined) ? this.data.toString() : '';
  return type + message + ' [' + data + ']';
}
/**
로그 스트림에 문자열을 출력합니다.
@param {string} message
*/
function log(message) {
  var logStream = document.getElementById('HandyLogStream');
  logStream.value += (message + '\n');
}
```

script 태그 내에 src 속성을 추가하여 handy.js 소스 파일을 포함하였다. 참고로 HTML 문서의 주석은 JavaScript의 주석과 달리 “<!--”, “-->” 기호를 이용한다. 이렇게 파일을 분리함으로써 HTML 문서의 크기가 줄고 필요한 부분만 볼 수 있게 되어 가독성이 높아지게 된다.

이후의 내용에서는 다시 이에 관해 얘기하기 전까지 handy.js를 기본으로 포함하고 위의 HTML 문서를 기본 형식으로 하자. 아직 우리의 로그 스트림은 개선해야 하는 부분이 더 있지만, 이는 문제가 닥칠 때마다 해결하는 것으로 하겠다.

5.3) StringBuffer

우리는 1장과 2장을 통해 식별자를 획득하는 방법을 이미 배웠다. 식별자를 획득하려면 왼쪽부터 시작하여 식별자의 조건에 맞는 문자를 모두 획득해야 한다. C 기준에서는 밑줄(_)이나 알파벳으로 시작하는 단어를 밑줄, 알파벳 또는 숫자가 아닌 문자가 나타날 때까지 문자를 획득해야 한다. 그리고 이 과정을 편리하게 하기 위해 StringBuffer 클래스를 작성하고 활용했었다.

여기서는 StringBuffer 클래스를 JavaScript를 이용하여 다시 작성한다. 가장 마지막으로 StringBuffer를 작성했던 프로젝트에 있는 StringBuffer 헤더 파일을 보이겠다.

StringBuffer.h

```
#ifndef __HANDY_STRINGBUFFER_H__
#define __HANDY_STRINGBUFFER_H__

#include <string>
#include "common.h"

typedef Exception StringBufferException;

class StringBuffer {
    std::string str;
    unsigned len;
    unsigned idx;
```

```
public:
//    explicit StringBuffer(const char *s = "");
    explicit StringBuffer(const std::string &str);
    ~StringBuffer();

    // 버퍼를 문자열로 초기화합니다.
    void init(const char *str);
    void init(const std::string &str);

    // 버퍼로부터 문자를 하나 읽습니다. 포인터가 이동합니다.
    char getc();
    // 버퍼의 포인터가 가리키는 문자를 가져옵니다. 포인터는 이동하지 않습니다.
    char peekc() const;
    // 버퍼에서 읽었던 값을 되돌립니다. 되돌릴 수 없으면 false를 반환합니다.
    bool ungetc();

    // 버퍼의 끝에 문자 또는 문자열을 추가합니다.
    void add(char c);
    void add(const char *s);
    void add(const std::string &str);

    // 버퍼가 비어있다면 true, 값을 더 읽을 수 있다면 false를 반환합니다.
    bool is_empty() const;

    // 버퍼로부터 정수를 획득합니다.
    std::string get_number();
    // 버퍼로부터 식별자를 획득합니다.
    std::string get_identifier();
    // 버퍼로부터 C 연산자를 획득합니다.
    std::string get_operator();
    // 공백이 아닌 문자가 나올 때까지 포인터를 옮깁니다.
    void trim();
    // 현재 위치 다음에 존재하는 토큰을 획득합니다.
    std::string get_token();
};

#endif
```

그럼 이를 바탕으로 StringBuffer 형식을 어떻게 구현해야할지를 고민해보자.

- #ifndef~

> StringBuffer 형식은 StringBuffer.js 파일로 분리하여 구현할 것이다. 이는 필요 없다.

- #include~

> JavaScript는 기본으로 문자열 형식을 지원하므로 string은 필요 없다. common에는 is_digit과 같은 기본 판별 함수가 들어가는데, String 객체가 이에 관한 메서드를 지원하지만 여기서는 별도로 만드는 것이 좋겠다. 즉 common.h와 common.cpp 파일을 바탕으로 common.js 파일을 작성한다. 추가로

말하자면 이전에는 예외 형식을 별도로 정의하는 대신 std::string 객체가 이 역할을 하도록 했었는데, 우리는 방금 Exception 객체에 관해 다루었던 만큼 앞으로 Exception 형식이 이를 대체하도록 할 것이다. Exception 형식은 handy.js에 정의했으므로 common에서 재정의 할 필요는 없다.

- typedef Exception StringBufferException;

> StringBuffer를 사용하면서 예외가 발생할 때 그 예외 정보를 담는 형식을 StringBufferException으로 정의한다. 이전에는 Exception 형식을 재정의 하는 수준에서 정의했지만, 여기서는 보다 디버깅할 때 도움이 될 정보를 추가할 것이다.

- class StringBuffer ~

> StringBuffer의 구현을 담고 있다.

그리고 이를 바탕으로 다음과 같이 StringBuffer.js를 구현한다.

5.3.1) 생성자와 초기화 함수

한 눈에 보기 힘들기 때문에 부분을 나누어 설명하겠다. 먼저 StringBuffer 생성자 함수의 정의는 다음과 같다.

StringBuffer.js (생성자 정의)

```
/**
  StringBuffer 생성자 함수를 정의합니다.
  @param {string} s
*/
function StringBuffer(s) {
  this.str = (s != undefined) ? s : '';
  this.idx = 0;
}
```

위에서 보인 StringBuffer.cpp 예제와 비교하면 len 멤버가 삭제되었음을 알 수 있다. JS의 문자열 객체는 기본적으로 length 멤버를 가지고 있기 때문에 굳이 멤버로 넣어야 할 필요가 없어 삭제했다. 사실 C++ 예제에서도 이 멤버는 삭제하는 것이 바람직하다.

생성자 정의에서 str 속성을 삼항 연산자를 이용하여 s로 정의하는 구문이 나온다. 처음에는 당황할 수 있지만, 잘 보면 그냥 s가 undefined인지 확인한 후 s가 undefined가 아니면 s를, undefined면 빈 문자열을 대입하는 문장임을 알 수 있다. 왜 그냥 this.str = s;와 같이 쓰지 않고 이런 식으로 썼을까?

이는 JavaScript에서 메서드를 호출하면 정의되지 않은 인자는 undefined가 된다는 특성에 의한 것이다. 다음 예제를 보자.

undefparam.htm

```
// 메서드의 인자를 출력합니다.
function func(param1, param2, param3) {
  log(param1);
  log(param2);
  log(param3);
}
function main() {
  func(1, 'test');  // func의 인자를 두 개만 넘긴다고 해도 적법한 문장입니다.
}
```

실행 결과

```
1
test
undefined
```

따라서 this.str = (s != undefined) ? s : '';를 수행하면 StringBuffer를 그냥 빈 문자열로 초기화하고 싶을 때 그냥 다음과 같이 하면 된다.

```
var buffer = new StringBuffer(); // StringBuffer('')처럼 쓸 필요가 없다
```

이제 생성자의 설명이 끝났으니 메서드를 보자. 다음은 버퍼를 초기화하는 init 메서드다.

StringBuffer.js (init)

```
/**
  버퍼를 문자열로 초기화합니다.
  @param {string} s
*/
StringBuffer.prototype.init = function(s) {
  this.str = (s != undefined) ? s : '';
  this.idx = 0;
};
```

생성자와 완전히 같은 코드다. 그런데 (s != undefined) ? s : '';와 같은 구문은 사실 꽤 자주 쓴다. 일단 undefined는 긴 키워드고, s에 들어갈 변수의 이름이 길면 우리는 다음과 같이 지루한 코드를 쓸 수밖에 없다.

```
var num = (value != undefined) ? value : 0;
var str = (valueString != undefined) ? valueString : '';
var isDragging = (isMouseButtonClicked != undefined) ? isMouseButtonClicked : false;
```

그러니 귀찮음을 덜기 위해 함수를 만들자. 다음 함수를 handy.js 파일에 추가한다.

handy.js (getValid)

```
/**
  value가 undefined라면 기본 값을, 아니면 그대로 반환합니다.
*/
function getValid(value, defaultValue) {
  return (value != undefined) ? value : defaultValue;
}
```

그러면 위의 문제는 다음과 같이 깔끔하게 해결이 된다.

```
var num = getValid(value, 0);
var str = getValid(valueString, '');
var isDragging = getValid(isMouseButtonClicked, false);
```

그리고 앞으로도 이러한 상황에서는 언제나 getValid 메서드를 사용할 것이다.

나머지는 사실상 코드를 복사 붙여넣기 하는 정도인데, 어느 부분이 달라지는지를 눈여겨보자.

5.3.2) 버퍼 기본 메서드: getc, peekc, ungetc, add, is_empty

다음은 StringBuffer의 확장 메서드를 구현하기 위해 필요한 getc 메서드를 구현한 것이다.

StringBuffer.js (getc)

```
/**
  버퍼로부터 문자를 하나 읽습니다. 포인터가 이동합니다.
  @return {string}
*/
StringBuffer.prototype.getc = function() {
  // 필드에 접근하기 위해 this 키워드를 반드시 붙여야 합니다.
  if (this.idx >= this.str.length)

    // throw와 StringBufferException 사이에 new가 붙었습니다.
    throw new StringBufferException('Buffer is empty', this);

  // 사실 StringBufferException의 두 번째 인자로 this도 넘겼습니다.
  return this.str[this.idx++];
}
```

이전 예제와 비교하여 세 가지가 달라졌다. 첫 번째는 필드에 접근하기 위해 반드시 this를 붙여야 한다는 것이다. 왜 그런지 살펴보자.

5.3.2.1) 왜 this가 붙는가?

this는 현재 함수가 호출된 위치에서 가장 가까운 객체에 접근하는 키워드다. 만약 this가 붙지 않는다면 idx와 str 같은 키워드를 가장 가까운 객체가 아닌 전역에서 찾는다. this는 이를 위한 것이다.

5.3.2.2) 왜 throw와 Exception 사이에 new가 붙는가?

다음으로 throw와 Exception 사이에 new가 붙는 것을 생각해보자. 이전 C++의 예제에서는 new를 붙여서 예외를 throw하지 않았다. C++의 new는 힙 메모리 영역에 동적으로 공간을 할당한 다음, 할당된 메모리의 주소를 반환하는 연산자다. 따라서 new 연산자로 할당한 메모리는 언제나 해제해주어야 하는데 이는 번거롭다. 그래서 이전 예제에서는 new 연산자를 이용하는 대신 이렇게 해결했었다.

TryCatch2.cpp

```
#include <iostream>

// 일반 예외 형식 Exception을 정의합니다.
class Exception {
    std::string _msg;
public:
    Exception(const std::string &msg) : _msg(msg) {}
    const char *toString() const { return _msg.c_str(); }
};

int main(void) {
    try {
        // 1. Exception("...") 구문이 임시 예외 객체를 생성합니다.
        // 2. 생성된 임시 예외 객체를 throw 합니다.
```

```
        throw Exception("예외가 발생했습니다.");
    }
    // 3. ex는 throw된 임시 예외 객체를 참조합니다.
    catch (Exception &ex) {
        // 4. 예외를 사용합니다.
        std::cout << "메시지: " << ex.toString() << std::endl;
        // 5. catch 블록을 벗어나면서 임시 객체에 대한 참조가 사라지고
        //  임시 예외 객체가 자동으로 소멸됩니다.
    }
    return 0;
}
```

주석에 나와 있듯, 먼저 임시 객체가 생성된 다음 이 예외 객체가 throw되고, catch 영역에서 이 예외를 사용한 다음 블록을 벗어나면 예외가 소멸되는 구조다. 이때 catch 영역에 참조를 뜻하는 &가 붙지 않으면 catch 영역에서 임시 객체를 catch할 때 ex가 throw된 임시 객체를 복사하게 된다. 그래서 성능에 저하가 발생하기 때문에 임시 예외 객체를 catch할 때는 &를 붙여 참조한다.

JavaScript에는 포인터라는 개념이 없다. 객체에 대한 모든 변수는 객체에 대한 참조다. C++만 배웠다면 이 말을 이해하기 쉽지 않을 테니 예를 들어보자. C++에서는 하나의 객체를 참조하기 위해서는 참조 또는 포인터를 사용한다.

RefObject.cpp

```
#include <iostream>

// Object 클래스를 정의합니다.
struct Object { int value; };

// 포인터를 이용하여 Object 객체의 값을 변경합니다.
void ChangeValue(Object *pObject, int value);
// 참조를 이용하여 Object 객체의 값을 변경합니다.
void ChangeValue(Object &rObject, int value);
// 인자는 사본이므로 이렇게는 값을 변경할 수 없습니다.
void ChangeValueWrong(Object o, int value);

int main(void) {
    Object object;

    // 객체가 선언된 지역이므로 값이 잘 변경됩니다.
    object.value = 10;
    std::cout << object.value << std::endl;

    // object의 사본이 생성되므로 값이 변경되지 않습니다.
    ChangeValueWrong(object, 20);
    std::cout << object.value << std::endl;

    // 포인터나 참조를 이용하면 값이 잘 변경됩니다.
```

```
    ChangeValue(&object, 30); // 포인터를 이용한 변경
    std::cout << object.value << std::endl;
    ChangeValue(object, 40); // 참조를 이용한 변경
    std::cout << object.value << std::endl;

    return 0;
}

// 포인터를 이용하여 Object 객체의 값을 변경합니다.
void ChangeValue(Object *pObject, int value) {
    pObject->value = value;
}
// 참조를 이용하여 Object 객체의 값을 변경합니다.
void ChangeValue(Object &rObject, int value) {
    rObject.value = value;
}
// 인자는 사본이므로 이렇게는 값을 변경할 수 없습니다.
void ChangeValueWrong(Object o, int value) {
    o.value = value;
}
```

실행 결과

```
10
10
30
40
```

반면 JS에서는 다음과 같이 코드를 작성하여 객체에 접근한다.

RefObjectJS.htm

```
// MyObject 형식을 정의합니다.
function MyObject(value) { this.value = value; }

// 값을 변경하는 함수를 정의합니다.
function changeValue(o, value) {
  // o는 object의 사본이 아닌 object에 대한 참조처럼 행동합니다.
  o.value = value;
}

function main() {
  var object = new MyObject();

  // object를 초기화하고 값을 출력합니다.
  object.value = 10;
```

```
  log(object.value);

  // 값을 변경하고 출력합니다.
  changeValue(object, 20);
  log(object.value);
}
```

각각의 changeValue 메서드를 자세히 보라. C++에서 함수의 인자는 객체에 대한 사본이고, JS에서는 객체에 대한 참조다. C++에서의 점 연산자는 현재 객체의 멤버에 접근하는 연산자고, JS에서의 점 연산자는 현재 변수가 가리키는 객체의 멤버에 접근하는 연산자로, 두 언어의 점 연산자가 역할이 다른 것이다. 정리하면 JS에는 포인터가 없고 변수는 객체에 대해 참조 정보를 갖는다.

결론은 throw와 Exception 사이에 new가 붙어 새로운 객체를 생성한 후 이 객체를 throw하면, catch 영역에서 이를 받는 ex는 throw된 Exception 객체에 대한 참조라는 것이다. 말이 아주 혼란스러우므로 한 번만 더 정리하자.

1. new Exception("...")을 통해 새로운 객체를 생성한다.
2. throw를 통해 생성한 Exception 객체를 throw한다. 이 객체를 thrownEx라고 하자.
3. catch 영역에서 throw된 Exception 객체를 받는다. 이는 ex = thrownEx이 실행된 것과 같다.

throw와 Exception 사이에 new가 들어가는 이유는 이 정도로 정리할 수 있다.

5.3.2.3) 왜 Exception의 인자로 this를 넘겼는가?

단순하다. this는 예외가 발생한 StringBuffer 객체를 말하는데, 이전에 StringBufferException에 예외에 대한 정보도 전달하겠다고 했다. 그럼 예외에 대한 정보는 당연히 StringBuffer 객체가 가지고 있을 것이다. 그래서 StringBufferException 객체에 예외에 대한 정보를 가지고 있는 StringBuffer 객체를 넘겼다. 이 정도면 설명할 수 있는 가장 친절한 선에서 설명한 것이라고 생각한다.

나머지는 peekc, ungetc와 같은 기본 메서드에 대한 내용인데 C++의 예제와 차이가 없으니 코드를 복사하는 것으로 마무리하겠다.

StringBuffer.js (peekc, ungetc, add, is_empty)

```
/**
  버퍼의 포인터가 가리키는 문자를 가져옵니다. 포인터는 이동하지 않습니다.
  @return {string}
*/
StringBuffer.prototype.peekc = function() {
  if (this.idx >= this.str.length)
    throw new StringBufferException('Buffer is empty', this);
  return this.str[this.idx];
}
/**
  버퍼에서 읽었던 값을 되돌립니다. 되돌릴 수 없으면 false를 반환합니다.
  @return {boolean}
*/
StringBuffer.prototype.ungetc = function() {
  if (this.idx > 0) {
    --this.idx;
    return true;
```

```
  }
  return false;
}

/**
  버퍼의 끝에 문자열을 추가합니다.
  @param {string} s
*/
StringBuffer.prototype.add = function(s) {
  this.str += s;
}

/**
  버퍼가 비어있다면 true, 값을 더 읽을 수 있다면 false를 반환합니다.
  @return {boolean}
*/
StringBuffer.prototype.is_empty = function() {
  return (this.idx >= this.str.length);
}
```

5.3.3) 버퍼 확장 메서드: get_number, get_identifier, get_operator, get_token

사실 우리는 이전 절에서 기본 메서드를 다루면서 JS에서의 특이 사항을 모두 학습했고, C++에서 작성했던 확장 메서드의 논리가 변하는 것도 아니기 때문에 변하는 부분을 눈으로만 확인하면 된다.

StringBuffer.js (get_number, get_identifier, get_operator)

```
/**
  버퍼로부터 정수를 획득합니다.
  @return {string}
*/
StringBuffer.prototype.get_number = function() {
  this.trim(); // 공백 제거
  if (this.is_empty()) // 버퍼에 남은 문자가 없다면 예외
    throw new StringBufferException("Buffer is empty", this);
  else if (is_digit(this.str[this.idx]) == false) // 첫 문자가 숫자가 아니면 예외
    throw new StringBufferException("invalid number", this);
  var value = '';
  while (this.is_empty() == false) {
    if (is_digit(this.str[this.idx]) == false)
      break;
    value += this.str[this.idx];
    ++this.idx;
  }
  return value;
}
```

```
/**
  버퍼로부터 식별자를 획득합니다.
  @return {string}
*/
StringBuffer.prototype.get_identifier = function() {
  this.trim(); // 공백 제거
  if (this.is_empty()) // 버퍼에 남은 문자가 없다면 예외
    throw new StringBufferException("Buffer is empty", this);
  else if (is_fnamch(this.str[this.idx]) == false)
    throw new StringBufferException("invalid identifier", this);
  var identifier = '';
  while (this.is_empty() == false) {
    if (is_namch(this.str[this.idx]) == false) // 식별자 문자가 아니라면 탈출
      break;
    identifier += this.str[this.idx];
    ++this.idx;
  }
  return identifier;
}

/**
  버퍼로부터 C 연산자를 획득합니다.
  @return {string}
*/
StringBuffer.prototype.get_operator = function() {
  this.trim();
  if (this.is_empty())
    throw new StringBufferException("Buffer is empty", this);
  var ch = this.str[this.idx++]; // 현재 문자를 획득하고 포인터를 이동한다
  var op = '';
  switch (ch) {
  case '+': op = ch; break;
  case '-': op = ch; break;
  case '*': op = ch; break;
  case '/': op = ch; break;
  default: throw new StringBufferException("invalid operator", this);
  }
  return op;
}

/**
  공백이 아닌 문자가 나올 때까지 포인터를 옮깁니다.
*/
```

```
StringBuffer.prototype.trim = function() {
    while (this.is_empty() == false) { // 버퍼에 문자가 남아있는 동안
        if (is_space(this.str[this.idx]) == false) // 공백이 아닌 문자를 발견하면
            break; // 반복문을 탈출한다
        ++this.idx; // 공백이면 다음 문자로 포인터를 넘긴다
    }
}
```

그리고 이 예제에서 get_token 메서드에 try-catch 구문이 추가된다.

StringBuffer.js (get_token)

```
/**
  현재 위치 다음에 존재하는 토큰을 획득합니다.
  토큰 획득에 실패하면 null을 반환합니다.
  @return {string}
*/
StringBuffer.prototype.get_token = function() {
  try {
    this.trim();
    if (this.is_empty())
        throw new StringBufferException("Buffer is empty", this);

    var ch = this.str[this.idx];
    var ss = ''; // 문자열 스트림 생성
    if (is_digit(ch)) { // 정수를 발견했다면 정수 획득
        ss += this.get_number(); // cout 출력 스트림처럼 사용하면 된다
    }
    else if (is_fnamch(ch)) { // 식별자 문자를 발견했다면 식별자 획득
        ss += this.get_identifier();
    }
    else { // 이외의 경우 일단 연산자로 획득
        ss += this.get_operator();
    }
    return ss; // 획득한 문자열을 반환한다
  } catch (ex) {
    // 토큰 획득에 실패한 경우 null을 반환합니다.
    return null;
  }
}
```

논리가 같다고 해도 변하는 부분이 없는 것은 아니므로 다른 부분을 확인하면서 이전 절에서 다룬 내용을 모두 이해하기 바란다.

5.3.4) StringBufferException

StringBufferException은 StringBuffer의 내부에서 발생하는 예외를 처리하기 위해 고안되었다. 코드를 먼저 보자.

StringBuffer.js (StringBufferException)

```
/**
StringBuffer의 예외 형식인 StringBufferException을 정의합니다.
@param {string} msg
@param {StringBuffer} data
*/
function StringBufferException(msg, data) {
  this.description = msg;
  this.data = data;
}

// JavaScript에는 문법적으로 상속 기능이 없기 때문에
// prototype 객체를 이용하여 상속을 흉내냅니다.
StringBufferException.prototype = new Exception();

// toString 메서드를 오버라이드 합니다.
StringBufferException.prototype.toString = function() {
  // 상위 객체의 메서드를 호출하고 반환된 문자열 앞에 'StringBuffer'를 붙입니다.
  return 'StringBuffer' + Exception.prototype.toString.call(this);
}
```

주석이 많아서 긴 코드처럼 보이지만 실제로는 8줄밖에 안 되는 짧은 코드다. 이 짧은 코드에도 배울 점이 있는데 바로 JavaScript에서의 상속에 대한 것이다. JavaScript에는 기본적으로 상속이 제공되지 않기 때문에 prototype을 이용하여 상속을 흉내 내는 방법을 설명하고자 하는 것이다.

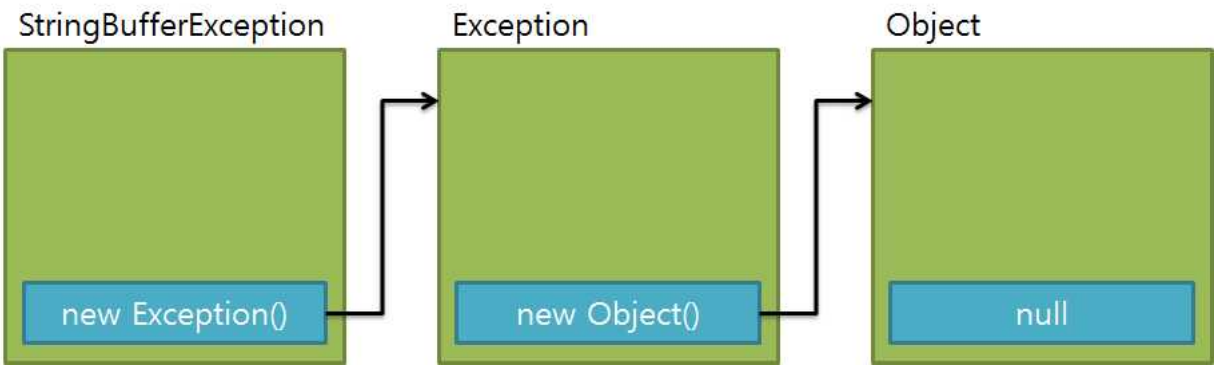
보통 코드를 설명할 때는 위에서 아래로 설명하는 편이지만, 여기서는 가운데 코드를 먼저 설명하겠다. 네 번째 줄을 먼저 보자.

```
StirngBufferException.prototype = new Exception();
```

6장에서도 말했고 방금도 말했는데, JS에는 상속이 없다. 모든 것은 객체고 생성자 함수가 클래스를 흉내 낸다. 그리고 이를 설명하면서 프로토타입 체인에 관해서도 얘기했다. 이 설명을 다시 가져와보자.

JavaScript에서 속성이나 메서드를 참조하게 되면, 먼저 자신 안에 멤버가 정의되어있는지 찾아본 다음, 발견하지 못하면 그 프로토타입으로 이동하여 해당 프로토타입 객체 내에서 멤버를 찾는다. 이는 멤버를 찾거나, 멤버를 찾지 못하고 null을 반환하고서야 비로소 끝나는데, 이러한 객체들의 연쇄를 가리켜 **프로토타입 체인(prototype chain)**이라고 한다.

이를 그림으로 표현하면 다음과 같다.



이 내용은 다음의 코드를 실행함으로써 확인할 수 있다.

StringBufferPrototype.htm

```
// 프로토타입을 확인하는 예제입니다.
function main() {
  var ex = new StringBufferException('test', null);

  // ex 객체의 프로토타입을 획득합니다.
  var p = Object.getPrototypeOf(ex);

  // 획득한 프로토타입 객체를 출력합니다.
  log(p);

  // 과정을 반복합니다.
  p = Object.getPrototypeOf(p);
  log(p);
  p = Object.getPrototypeOf(p);
  log(p);
  p = Object.getPrototypeOf(p);
  log(p);
  log('complete');
}
```

실행 결과

```
StringBufferException: undefined []
Exception: undefined []
[object Object]
null
complete
```

이 경우 생성한 StringBufferException 객체는 내부에 그 상위 객체 Exception의 인스턴스를 가지고 있다. 같은 방식으로 Exception 또한 내부에 Object의 인스턴스를 가지고 있고, 모든 객체의 조상인 Object 객체만이 유일하게 상위 객체가 없이 null로 초기화되어있다.

사실 JavaScript의 상속은 이 내용만으로는 모든 것이 설명되지 않았지만, 일단 프로젝트를 진행하기 위해서는 이 정도로 충분한 것 같다. 후에 기회가 된다면 이에 대해서도 별도의 문서에서 다루겠다.

나머지는 메서드 오버라이딩에 관한 내용이다. 어렵지 않게 이해할 수 있으므로 넘어가겠다.

5.4) HandyFileSystem

Runner는 어셈블리 코드를 번역한 파일을 내놓는다. 따라서 파일 시스템에 접근해야 한다. 그런데 서문에서 브라우저에 내장된 JavaScript를 이용해서는 파일 시스템에 접근할 수 없다는 사실을 이미 밝혔다. 따라서 우리는 파일 시스템에 접근할 수 있도록 Chrome 브라우저 대신 node.js 프로그램을 이용하여 개발을 진행해야 한다. brackets는 편한 개발도구이므로 그대로 사용하지만, 사용 방법은 약간 변한다. 이에 대해서는 블로그에 brackets에서 nw.js를 사용하는 방법을 설명하는 포스트[2]를 작성했으니 참고하기 바란다.

2) http://hdnua.tistory.com/

이 절에서는 파일 시스템에 접근하기 위한 래퍼 객체인 HandyFileSystem 싱글톤 객체를 작성한다. 이는 자주 사용하는 객체이므로 handy.js 파일에 구현할 것이다. 그런데 이를 만드는 방법이 이전과 약간 다르니 주의해서 코드를 봐야 한다.

먼저 HandyFileSystem 객체를 초기화하는 initHandyFileSystem 함수를 부분을 나눠서 보자.

handy.js (initHandyFileSystem)

```
/**
  HandyFileSystem 싱글톤 객체를 생성하고 초기화합니다.
*/
function initHandyFileSystem() {

  // HandyFileSystem 싱글톤 객체를 정의합니다.
  // 빈 객체를 먼저 생성함에 주의합니다.
  var hfs = {};

  ... // HandyFileSystem 객체의 속성을 구현합니다.

  // 전역 객체를 의미하는 window 객체에 HandyFileSystem 속성을 추가하고
  // 생성한 HandyFileSystem 싱글톤 객체를 대입합니다.
  window['HandyFileSystem'] = hfs;
}
```

HandyFileSystem이라는 객체를 정의한다면서 var HandyFileSystem = { ... };과 같은 식이나 생성자 함수를 정의하지 않고, initHandyFileSystem이라는 함수를 만든 다음 함수 내에 지역 변수로 빈 객체를 생성했다. 그리고 함수의 마지막 부분에 window['HandyFileSystem'] = hfs;라는 이상한 문장을 수행한다. 이것이 어떻게 싱글톤 객체의 정의가 될 수 있을까?

이 이야기를 진행하기 전에 아직 설명하지 않은 window라는 녀석에 대해 먼저 설명해야겠다. JS에는 문서가 기본적으로 지원하는 객체가 있다. textarea 요소에 접근하기 위해 document.getElementById라는 메서드를 호출했던 것을 기억하는가? 이때 getElementById는 document라는, JS가 기본으로 제공하는 객체가 가진 메서드다. 이런 기본 객체는 window, document 말고도 여러 가지가 있지만 이에 대해서 이 문서에서 자세히 다루지는 않겠다. 중요한 건 window와 document다.

window는 JS 코드 내에서 생성되는 모든 객체에 대한 기본 객체다. Object는 모든 객체에 대한 기본 형식이라고 하면, window는 생성한 객체에 대한 기본 영역이 된다. 여전히 말을 이해하지 못하겠다면 다음의 예제를 보라.

window.htm <html.head.script>

```
// main 함수의 바깥에서 num과 func를 정의합니다.
// 이때 num과 func가 정의된 영역을 '전역'이라고 하면,
var num = 10;
function func() {
  log('testfunction');
}

function main() {
  // 전역에 정의된 멤버는 모두 window 객체의 속성이 됩니다.
```

```
  log(window.num);
  log(window.func);

  // 특별히 소속이 없는 모든 속성과 메서드는 window 객체의 속성입니다.
  window.alert('hello, world!');

  // window 객체의 멤버처럼 alert 메서드를 호출할 수 있습니다.
  window['alert']('this is also possible');
}
```

실행 결과 (순서대로 로그, 경고1, 경고2)

```
10
function func() {
        log('testfunction');
      }
```

```
hello, world!
```

```
this is also possible
```

이와 같이 전역에 정의된 임의의 변수, 함수 또는 객체는 모두 window 객체의 속성이 된다. 사실 전역에 변수나 함수, 객체를 정의하는 행위는 다음을 수행하는 것과 같다.

```
window.num = 10;
window['func'] = function() { /* */ };
window['person'] = { name:'handy', age:20 };
```

따라서 이러한 특징을 이용하여 HandyFileSystem 속성을 window 객체에 추가하였다. 이에 따라 우리는 임의의 위치에서 HandyFileSystem에 접근하는 것이 가능해졌다. 잠시 후에 이에 대한 예제를 보일 것이다.

이제 HandyFileSystem 객체의 구현을 보자.

handy.js (initHandyFileSystem)

```
function initHandyFileSystem() {
  ...
  var hfs = {};

  // 파일 시스템 객체를 획득합니다.
  var fso = require('fs');

  // 현재 작업중인 파일의 디렉터리를 획득합니다.
  var gui = require('nw.gui');
  var dir = gui.App.argv[0];

  ...

  // 정의한 속성을 HandyFileSystem 싱글톤 객체의 멤버로 정의합니다.
```

```
  hfs.fso = fso;
  hfs.dir = dir;
  ...
}
```

이 부분에서는 fso, gui, dir이라는 세 변수를 생성하였다. 이들에 대해 간단히 설명하겠다.

fso는 파일 시스템에 접근하기 위해 필요한 모듈이다. gui는 현재 경로를 얻기 위해 gui.App.argv 속성에 접근해야 해서 가져온 모듈이다. 그 외의 용도로는 사용하지 않는데, 이에 대해 자세한 내용을 알고 싶다면 필자의 블로그[3]를 참조하라. 이를 이용해 얻은 경로는 dir 변수에 저장한다.

그리고 마지막에 hfs.fso = fso;와 같이 값을 대입하는 식이 나오는데, 이는 기존 객체에 멤버를 추가하는 것이다. 6장에서 이에 대해 설명했으니 기억이 나지 않는다면 다시 찾아보길 바란다.

마지막으로 HandyFileSystem의 메서드를 설명하겠다. 먼저 load 함수를 보자.

handy.js (initHandyFileSystem)

```
function initHandyFileSystem() {
  ...

  /**
  // 파일에 기록된 텍스트를 모두 불러와 문자열로 반환합니다.
  // 성공하면 획득한 문자열, 실패하면 null을 반환합니다.
  @param {string} filename
  @return {string}
  */
  function load(filename) {
    try {
      var filepath = this.dir + '\\' + filename;
      return this.fso.readFileSync(filepath);
    } catch (ex) {
      return null;
    }
  }

  ...

  // 정의한 속성을 HandyFileSystem 싱글톤 객체의 멤버로 정의합니다.
  ...
  hfs.load = load;
  ...
}
```

JavaScript에서는 함수 내부에 함수를 정의하는 것이 가능하므로 load 함수를 정의했다. 중요한 특징인데, JavaScript에서 함수 내부에서, 즉 지역에서 함수를 정의한다고 전역에서도 바로 이 함수를 호출할 수는 없다. 지역에서 정의된 함수는 지역적인 성질을 가진다. 이는 다음의 예제에서 확인할 수 있다.

3) http://hdnua.tistory.com/15

localfunc.htm

```
// 전역 함수 gfunc를 정의합니다.
function gfunc() {
  // 지역적인 함수 lfunc를 정의합니다.
  function lfunc() {
    alert('hello, world!');
  }

  // lfunc의 내용이 잘 출력됩니다.
  log(lfunc);
  // lfunc가 전역에 정의되지 않았으므로 undefined가 출력됩니다.
  log(window.lfunc);
}

function main() {
  gfunc();
}
```

실행 결과

```
function lfunc() {
    alert('hello, world!');
  }
undefined
```

load 함수 내부는 fso 객체의 메서드 readFileSync를 이용하여 파일에 있는 모든 텍스트를 가져온 다음 문자열로 반환하는 단순한 코드로 이루어져있다. 실패하면 null을 반환한다. 마지막으로 hfs의 load 속성을 우리가 정의한 지역 함수 load로 맞춘다. 이게 끝이다.

나머지는 방식이 완전히 똑같으므로 코드만 보이겠다.

handy.js (initHandyFileSystem)

```
function initHandyFileSystem() {
  ...

  /**
  // 파일에 텍스트를 기록합니다.
  // 기록에 성공하면 true, 실패하면 false를 반환합니다.
  @param {string} filename
  @return {Boolean}
  */
  function save(filename, data) {
    try {
      var filepath = this.dir + '\\' + filename;
      this.fso.writeFileSync(filepath, data);
```

```
      return true;
    } catch (ex) {
      return false;
    }
  }
  /**
  // 파일이 디스크에 존재하는지 확인합니다.
  @param {string} filepath
  @return {Boolean}
  */
  function exists(filepath) {
    return this.fso.exists(filepath);
  }

  ...

  // 정의한 속성을 HandyFileSystem 싱글톤 객체의 멤버로 정의합니다.
  ...
  hfs.save = save;
  hfs.exists = exists;
  ...
}
```

결국 initHandyFileSystem은 다음과 같은 식으로 구성되어있다.

handy.js (initHandyFileSystem)

```
function initHandyFileSystem() {
  // 1. 빈 객체 생성
  var hfs = {};

  // 2. 필드와 메서드 정의
  var fso = ..., dir = ...;
  function load(...) { ... }
  function save(...) { ... }
  function exists(...) { ... }

  // 3. 빈 객체에 속성 추가
  hfs.fso = fso; hfs.dir = dir;
  hfs.load = load; hfs.save = save; hfs.exists = exists;

  // 4. 전역에 작성한 hfs 객체 등록
  window['HandyFileSystem'] = hfs;
}
```

앞으로 만들 모든 싱글톤 객체는 거의 이런 식으로 구성할 것이므로, 이 형태에 익숙해지기 바란다. 이로써 HandyFileSystem에 대한 설명이 끝났다. 다음은 이를 활용하는 코드다.

main.html <html.head.script>

```
// HandyFileSystem demonstration
function main() {

  // HandyFileSystem 객체를 가리키는 변수입니다.
  var hfs = HandyFileSystem;

  // 파일에 문장을 기록하는 예제입니다.
  var filename = 'handymakesman.txt';
  var text = prompt('Enter text to write', 'hello, world!');
  if (hfs.save(filename, text) == false)
    log('failed');
  else
    log('complete');

  // 파일에 기록된 텍스트를 불러오는 예제입니다.
  var text = hfs.load(filename);
  if (text) {
    log('succeed: ' + text);
  }
  else {
    log('failed to load');
  }
}
function init() {
  var logStream = document.getElementById('HandyLogStream');
  logStream.style.width = '100%';
  logStream.style.height = '100%';
  initHandyFileSystem();
}
```

실행 결과

```
complete
hello, world!
```

이와 같이 HandyFileSystem을 정의할 수 있었다.

6. 단원 마무리

원래는 이 문서에 어셈블리 실행기에 대한 내용을 넣으려고 했다. 그런데 일단 페이지가 지나치게 많아지는 점도 있고, 문서를 나누는 게 읽고 관리하기 좋다는 점, 오늘 내일 안에 다 쓰기엔 좀 무리가 있을 것 같다는 점(=_=) 때문에 여기서 한 번 필요한 내용만 모으는 것이 낫다고 생각했다.

다음 문서에서는 4절에서 보였던 HASM 코드를 실행하는 어셈블리 실행기인 Runner 모듈을 개발한다. Runner는 정말 재미있는 모듈이다. 이를 공부하면서 프로세스가 어떻게 실행되는지에 대해 깊이가 달라질 것이라고 생각한다.