

Korea University MatKor 스터디 – C 컴파일러 만들기

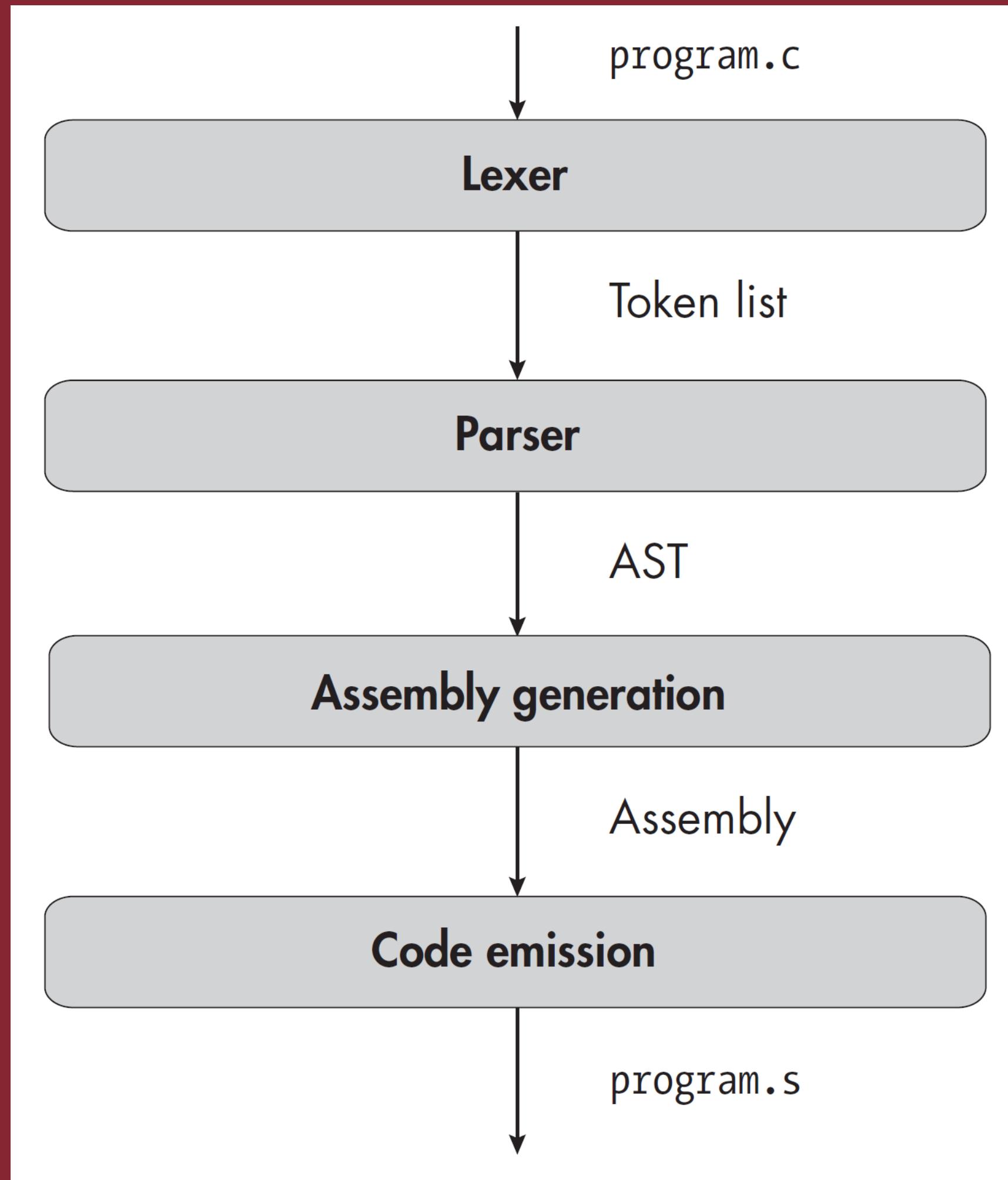
# A Minimal Compiler

Chris Ohk

utilForever@gmail.com

# 들어가며

Korea University MatKor 스터디  
A Minimal Compiler



# The Four Compiler Passes

Korea University MatKor 스터디  
A Minimal Compiler

- **Lexer**
  - 소스 코드를 토큰 리스트로 분해
  - 토큰은 프로그램의 가장 작은 구문 단위로, 키워드, 구분자, 식별자, 상수 등을 포함한다.
  - 프로그램이 책이라면, 토큰은 개별 단어와 같다.
- **Parser**
  - 토큰 리스트를 **AST(Abstract Syntax Tree)**로 변환
  - 쉽게 탐색하고 분석할 수 있는 형태로 프로그램을 표현한다.
- **Assembly Generation**
  - AST를 어셈블리로 변환
  - 어셈블리 명령어를 텍스트 형태가 아닌 컴파일러가 이해할 수 있는 자료 구조로 표현한다.
- **Code Emission**
  - 어셈블리 코드를 파일에 기록해 어셈블러와 링커가 이를 실행 파일로 변환할 수 있도록 한다.

# The Four Compiler Passes

---

Korea University MatKor 스터디  
A Minimal Compiler

- 컴파일러를 구성하는 전형적인 방식이지만, 정확한 단계와 중간 표현은 달라질 수 있다.
- 이후 더 많은 언어 기능을 구현하면서 컴파일러 단계를 확장하고  
몇 가지 새로운 단계가 추가될 예정이다.

# Hello, Assembly!

Korea University MatKor 스터디  
A Minimal Compiler

- 간단한 C 프로그램을 생각해 보자.

```
● ● ●  
  
int main(void) {  
    return 2;  
}
```

- return\_2.c로 저장한 후 다음 gcc 명령어를 사용해 어셈블리로 변환해 보자.

```
● ● ●  
  
gcc -S -O -fno-asynchronous-unwind-tables -fcf-protection=none return_2.c
```

# Hello, Assembly!

Korea University MatKor 스터디  
A Minimal Compiler

- 명령줄 옵션
  - -S : 어셈블러나 링커를 실행하지 않게 한다. 컴파일러가 바이너리 파일 대신 어셈블리 코드를 생성하도록 한다.
  - -O : 코드를 최적화한다. 이 명령은 현재 신경쓰지 않는 명령어 중 일부를 제거한다.
  - -fno-asynchronous-unwind-tables : 디버깅에 사용되는 Unwind Table을 생성하지 않는다.
  - -fcf-protection=none : 제어 흐름 보호 기능을 비활성화한다. 이 보호 기능은 신경쓰지 않는 명령어들을 추가한다.
- 결과로 `return_2.s`이 생성되며 그 내용은 다음과 같다.

```
● ● ●

.global main
main:
    movl    $2, %eax
    ret
```

# The Compiler Driver

Korea University MatKor 스터디  
A Minimal Compiler

- Lecture 0에서 설명했듯이, 컴파일러 자체만으로는 완성되지 않는다.  
전처리기, 컴파일러, 어셈블러, 링커를 호출하는 컴파일러 드라이버가 필요하다.
- 따라서 컴파일러를 작성하기 전에 컴파일러 드라이버를 먼저 작성하게 된다.  
이 드라이버는 소스 파일을 실행 파일로 변환하는 3단계를 거쳐야 한다.
  1. 소스 파일을 전처리하기 위해 다음 명령을 실행한다.

```
● ● ●  
gcc -E -P INPUT_FILE -o PREPROCESSED_FILE
```

2. 전처리된 소스 파일을 컴파일하고 .s 확장자를 가진 어셈블리 파일을 출력한다.
3. 다음 명령을 실행해 어셈블리 파일을 어셈블하고 링크해 실행 파일을 생성한다.

```
● ● ●  
gcc ASSEMBLY_FILE -o OUTPUT_FILE
```

# The Lexer

Korea University MatKor 스터디  
A Minimal Compiler

- 소스 파일을 읽어 토큰 목록을 생성해야 한다.  
렉서 작성을 시작하기 전에, 어떤 토큰들을 만날 수 있는지 알고 있어야 한다.

<b>int</b>	A keyword
<b>main</b>	An identifier, whose value is “main”
(	An open parenthesis
<b>void</b>	A keyword
)	A close parenthesis
{	An open brace
<b>return</b>	A keyword
<b>2</b>	A constant, whose value is “2”
;	A semicolon
}	A close brace

- **Identifier(식별자)** : ASCII 문자, 밑줄(\_)로 시작하는 문자, 밑줄, 숫자의 조합
- 정수 **Constant(상수)** : 하나 이상의 숫자로 구성

# The Lexer

- 토큰 목록 중 식별자와 상수는 값을 가지고 있다는 특징이 있다.
  - 식별자 토큰은 이름을 저장해야 한다. (예 : foo, variable1, my\_cool\_function 등)
  - 상수 토큰은 정수 값을 저장해야 한다. (예 : 42, -123 등)
- 각 토큰 타입은 정규 표현식(Regular Expression)으로 인식할 수 있다.

Token	Regular expression
Identifier	[a-zA-Z_]\w*\b
Constant	[0-9]+ \b
int keyword	int\b
void keyword	void\b
return keyword	return\b
Open parenthesis	\(
Close parenthesis	\)
Open brace	{
Close brace	}
Semicolon	;

- 프로그램을 토큰화하는 과정은 다음과 같다.

```
while input isn't empty:  
    if input starts with whitespace:  
        trim whitespace from start of input  
    else:  
        find longest match at start of input for any regex in Table 1-1  
        if no match is found, raise an error  
        convert matching substring into a token  
        remove matching substring from start of input
```

- 렉서 구현 관련 몇 가지 팁

- 키워드를 다른 식별자와 똑같이 처리하라.
- 공백으로 분할하지 마라.
- ASCII 문자만 지원하면 된다.

# The Parser

---

- 토큰 목록을 확보했으니, 이제 이 토큰들이 어떻게 언어 구조체로 그룹화되는지 알아보자.
- C를 포함한 대부분의 프로그래밍 언어에서 이 그룹화는 계층적이다.
  - 프로그램 내 각 언어 구조체는 여러 단순한 구조체로 구성된다.
  - 각 토큰은 변수, 상수, 산술 연산자 같은 가장 기본적인 구조체를 나타낸다.
  - 이 계층적 관계를 표현하는 가장 자연스러운 방법은 트리 자료 구조다.
- 파서는 렉서가 생성한 토큰 목록을 받아 AST를 생성한다.
- 파서가 AST를 생성한 후에는 어셈블리 생성 단계에서 이를 탐색해 어떤 어셈블리 코드를 출력할지 결정한다.

# The Parser

---

Korea University MatKor 스터디  
A Minimal Compiler

- 파서를 작성하는 2가지 접근 방식
  - 직접 손으로 만든다.
  - Bison이나 ANTLR 같은 파서 생성기를 사용한다.
- 직접 손으로 만드는 이유
  - 파서가 어떻게 동작하는지에 대해 제대로 이해할 수 있기 때문이다.
  - 파서 생성기를 사용하면 생성된 코드를 완전히 이해하지 못한 채로 사용하기 쉽다.
  - 손으로 만든 파서는 생성기로 만든 파서보다 더 빠르고 디버깅이 쉬우며, 유연성이 높고 오류 처리를 더 잘 지원한다.

# An Example AST

Korea University MatKor 스터디  
A Minimal Compiler

- 다음 코드를 보자.



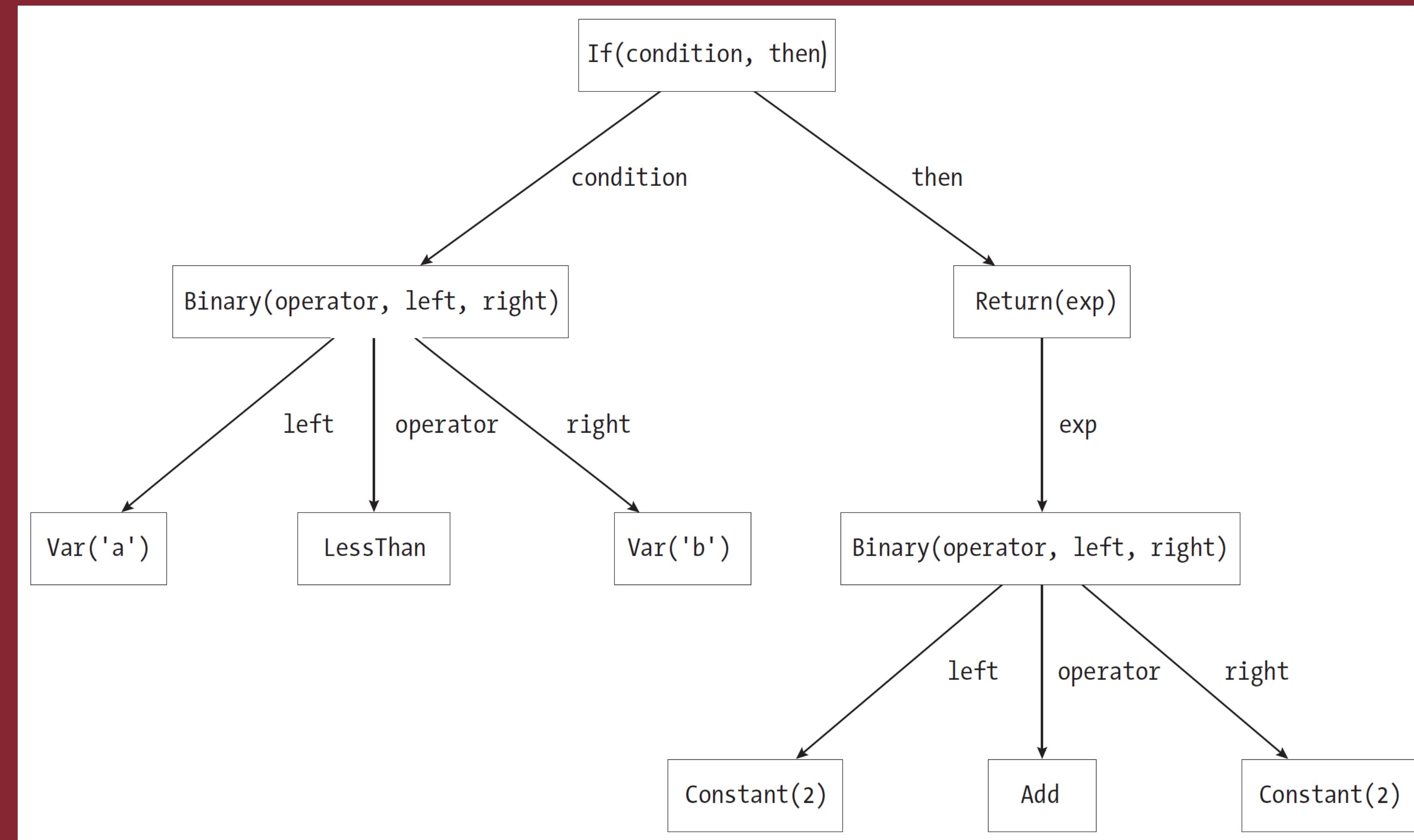
```
● ● ●  
if (a < b) {  
    return 2 + 2;  
}
```

A screenshot of a terminal window with a dark background. At the top, there are three colored dots: red, yellow, and green. Below them, the code for a simple if statement is displayed in white text. The code consists of the keyword 'if' followed by a condition '(a < b)', an opening brace '{', a single-line block containing 'return 2 + 2;', and a closing brace '}'.

- 해당 AST의 루트 노드는 if문 전체를 나타낸다.
- 이 노드에는 자식 노드가 2개 있다.
  - 조건 : a < b
  - 본문 : return 2 + 2;
- 각 자식 노드는 더 세분화될 수 있다. 예를 들어, 조건은 3개의 자식을 가진 이항 연산이다.
  - 왼쪽 피연산자 : 변수 a
  - 연산자 : <
  - 오른쪽 피연산자 : 변수 b

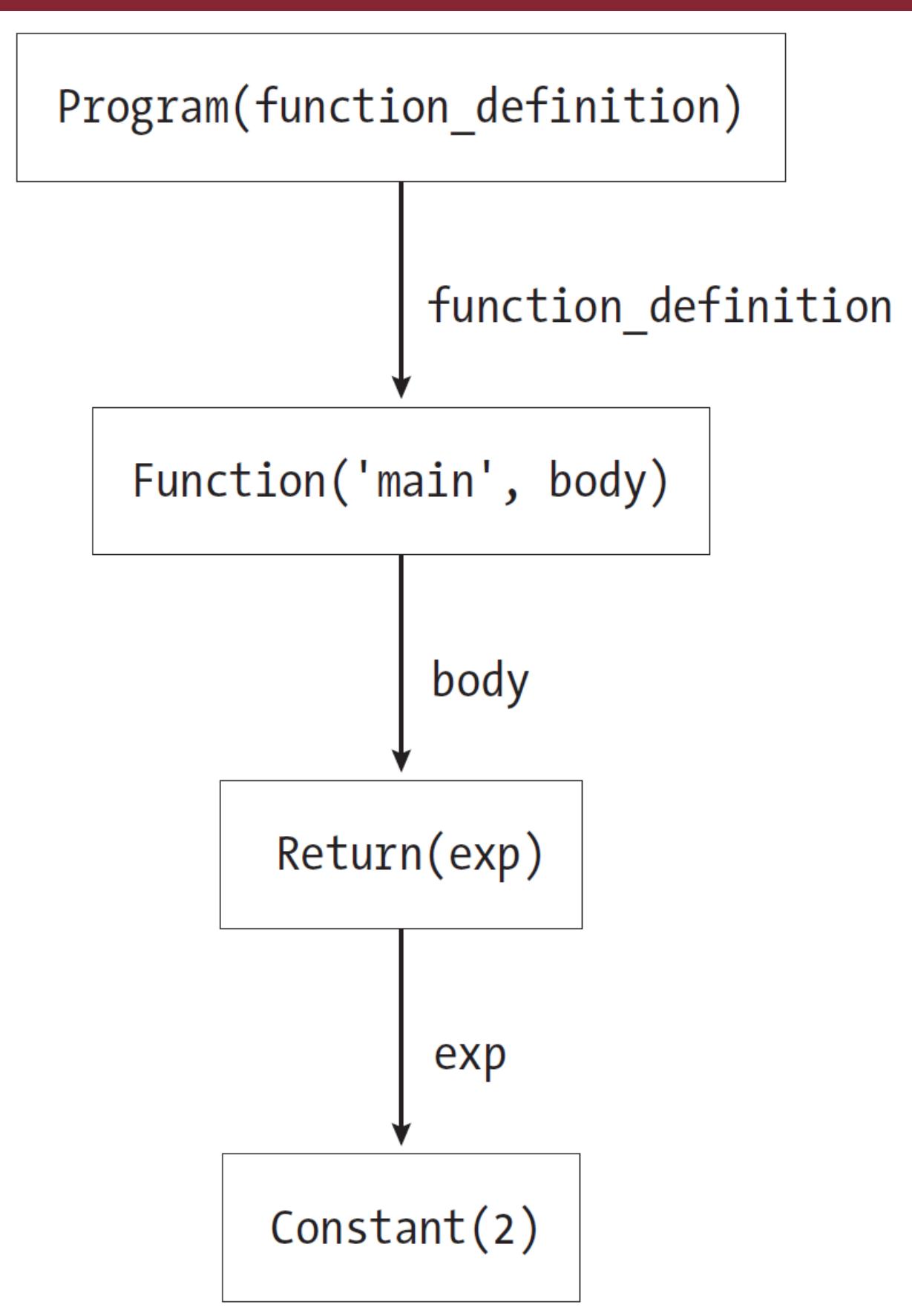
# An Example AST

- 이전 코드의 AST는 다음과 같다.



# An Example AST

- Hello, Assembly!에서 봤던 코드의 AST는 훨씬 단순하다.



# The AST Definition

Korea University MatKor 스터디  
A Minimal Compiler

- AST 설명을 Zephyr Abstract Syntax Description Language(ASDL)로 제공한다.
  - 여기에서는 ASDL에 대해 간단하게만 다룬다.  
자세한 내용은 <https://www.cs.princeton.edu/~appel/papers/asdl97.pdf>을 참고
  - C언어의 아주 작은 부분집합에 대한 ASDL 정의

```
program = Program(function_definition)
function_definition = Function(identifier name, statement body)
statement = Return(exp)
exp = Constant(int)
```

- 각 줄은 한 종류의 AST 노드를 생성하는 방법을 설명한다.
- 이 AST의 루트는 program 노드다. 이 노드는 정확히 1개의 자식 노드(function\_definition 타입)을 가질 수 있다.
- 함수 정의는 2개의 자식 노드, 함수 이름(identifier name)과 본문(statement body)을 가진다.
  - 현재 함수는 1개의 문장으로 구성되고 인수를 가지지 않는다. 이후 함수 인수와 더 복잡한 함수 본문을 위한 지원이 추가될 것이다.
  - name과 body는 필드 이름(Field Name)으로 AST 구조를 변경하지 않는 인간 친화적인 레이블이다.
  - 필드 이름은 ASDL에서 선택 사항이다. 필드 이름이 존재할 경우, identifier name과 같이 필드 타입 바로 뒤에 위치한다.

# The AST Definition

---

Korea University MatKor 스터디  
A Minimal Compiler

- C언어의 아주 작은 부분집합에 대한 ADSL 정의
  - ADSL에서 식별자(identifier)는 함수 및 변수 이름을 나타내는 내장 타입이다.
    - 기본적으로 문자열이지만, “Hello, WorlD!”와 같은 문자열 리터럴과 구분하기 위해 사용한다. (AST의 다른 부분에 나타나기 때문)
    - 내장 타입이므로 자식 노드가 없다.
  - 함수 정의 노드의 다른 자식 노드는 문(statement)이다.
    - 현재 유일한 문 타입은 반환(return)문이다. 반환문은 하나의 자식 노드 exp를 가진다. (expression의 줄임말)
    - 현재 유일한 표현식은 상수 정수인 int다. int는 또 다른 ADSL 내장 타입이므로 자식 노드가 없다.
  - 이렇게 AST는 완성되었다.

# The AST Definition

- C언어의 아주 작은 부분집합에 대한 ADSL 정의
  - 물론 C에는 반환문 외에 다양한 문들이 있고 상수 외에 다양한 표현식들이 있다. 이후 강의를 진행하며 다른 종류의 문과 표현식을 표현하기 위해 새로운 생성자를 추가할 것이다.
  - 예를 들어, if문을 표현하기 위해 statement에 다음과 같이 If 생성자를 추가할 것이다.

```
<statement> ::= "return" <exp> ";" | "if" "(" <exp> ")" <statement> [ "else" <statement> ]
```

- statement? 타입은 선택적 문장을 나타낸다. 왜냐하면 if문이 항상 else절을 가지는 건 아니기 때문이다.
- | 기호는 생성자를 구분한다.  
(Return 생성자로 정의된 return문이거나 If 생성자로 정의된 if문일 수 있음을 알려준다.)

# The Formal Grammar

Korea University MatKor 스터디  
A Minimal Compiler

- AST는 컴파일러의 후속 단계에서 필요한 모든 정보를 포함한다.  
그러나 각 언어 구문을 구성하는 토큰이 정확히 무엇인지는 알려주지 않는다.
- 예를 들어, 앞에서 봤던 프로그램의 AST 설명 어디에도 return문이 세미콜론으로 끝나야 한다거나  
함수 본문이 중괄호로 묶여 있어야 한다는 내용은 없다.
  - 추상(Abstract) 구문 트리인 이유가 바로 여기에 있다.
  - 반면 구체(Concrete) 구문 트리는 원본 입력의 모든 토큰을 포함한다.
- AST를 확보한 후에는 이런 세부 사항이 무의미해지므로 생략하는 것이 편리하다. 하지만 토큰 시퀀스를 파싱해서  
AST를 구성할 때는 각 언어 구문의 시작과 끝을 나타내므로 이런 세부 사항이 매우 중요하다.

# The Formal Grammar

Korea University MatKor 스터디  
A Minimal Compiler

- 따라서 토큰 목록으로부터 언어 구문을 생성하는 방법을 정의하는 규칙 집합이 필요하다.
  - 이 규칙 집합을 **형식 문법(Formal Grammer)**이라고 하며, AST 설명과 밀접하게 대응한다.
  - 다음은 앞에서 봤던 C 프로그램에 대한 형식 문법을 정의한다.

```
<program> ::= <function>
<function> ::= "int" <identifier> "(" "void" ")" " {" <statement> "}"
<statement> ::= "return" <exp> ";"
<exp> ::= <int>
<identifier> ::= ? An identifier token ?
<int> ::= ? A constant token ?
```

- 이 문법은 **EBNF(Extended Backus-Naur Form)** 표기법으로 작성되었다.  
각 행은 언어 구조가 다른 언어 구조와 토큰의 순서로 어떻게 형성될 수 있는지를 정의하는 **생성 규칙(Production Rule)**이다.
- 생성 규칙의 왼쪽에 나타나는 모든 기호(예 : <function>)은 **비종단 기호(Non-terminal Symbol)**다.  
키워드, 식별자, 구두점과 같은 개별 토큰은 **종단 기호(Terminal Symbol)**다.
- 모든 비종단 기호는 각괄호로 묶이며, 특정 토큰(예 : ;)은 따옴표로 묶인다.  
<identifier> 및 <int> 기호는 각각 개별 식별자 토큰과 상수 토큰을 나타낸다.
- 이러한 토큰은 다른 종단 기호처럼 고정된 문자열이 아니므로, 각 기호를 ?로 묶인 특별한 시퀀스로 나타낸다.

# The Formal Grammar

Korea University MatKor 스터디  
A Minimal Compiler

- 따라서 토큰 목록으로부터 언어 구문을 생성하는 방법을 정의하는 규칙 집합이 필요하다.
  - 이 규칙 집합을 **형식 문법(Formal Grammer)**이라고 하며, AST 설명과 밀접하게 대응한다.
  - 다음은 앞에서 봤던 C 프로그램에 대한 형식 문법을 정의한다.

```
<program> ::= <function>
<function> ::= "int" <identifier> "(" "void" ")" "{" <statement> "}"
<statement> ::= "return" <exp> ";"
<exp> ::= <int>
<identifier> ::= ? An identifier token ?
<int> ::= ? A constant token ?
```

- 이 형식 문법은 앞에서 봤던 AST 정의와 매우 유사하다. 사실 동일한 구조를 갖는다. 모든 AST 노드는 비종단 기호에 대응한다. 유일한 차이점은 형식 문법이 트리의 각 노드에서 정확히 어떤 토큰을 찾을지 명시한다는 점이다.
- 이는 AST에서 다음 하위 레벨의 새 노드 처리를 시작해야 할 시점과 노드 처리를 완료하고 상위 레벨의 부모 노드로 돌아갈 수 있는 시점을 파악하는 데 도움이 된다.

# The Formal Grammar

Korea University MatKor 스터디  
A Minimal Compiler

- 따라서 토큰 목록으로부터 언어 구문을 생성하는 방법을 정의하는 규칙 집합이 필요하다.
  - 앞에서 이후 강의를 진행하며 여러 생성자를 추가할 것이라고 안내했었다.  
해당 기호에 대한 여러 생성 규칙도 같이 추가할 것이다.
  - 예를 들어, if문을 지원하기 위한 <statement>에 대한 생성 규칙을 추가하는 방법은 다음과 같다.

```
<statement> ::= "return" <exp> ";" | "if" "(" <exp> ")" <statement> [ "else" <statement> ]
```

- EBNF에서 대괄호([ ])는 선택 사항이다. (ASDL에서 물음표가 선택 사항인 것과 동일하다.)

# Recursive Descent Parsing

Korea University MatKor 스터디  
A Minimal Compiler

- 이제 AST 정의와 형식 문법이 준비되었으니, 실제로 파서를 작성하는 방법을 이야기해 보자.

우리는 **재귀 하향 파싱(Recursive Descent Parsing)**이라는 기법을 사용할 것이다.

- 이 기법은 각 비종단 기호를 파싱하고 해당 AST 노드를 반환하기 위해 서로 다른 함수를 사용한다.
  - 예를 들어, 파서가 <statement> 기호를 만나면 해당 기호를 파싱하고 AST 노드 statement를 반환하는 함수를 호출한다.
  - 메인 파싱 함수는 전체 프로그램에 해당하는 <program> 기호를 파싱한다.
  - 새로운 기호를 처리하기 위한 함수 호출이 있을 때마다 파서는 트리에서 더 낮은 수준으로 하향한다.
  - 재귀 하향의 '하향'이 유래된 이유가 바로 여기에 있다.

문법 규칙이 종종 재귀적이기 때문에 함수도 재귀적으로 처리하므로 재귀 하향이라고 한다.

예를 들어, 표현식의 피연산자가 또 다른 표현식일 수 있다. (자세한 내용은 다음 강의에서 살펴보도록 하자.)

# Recursive Descent Parsing

- 파싱 함수 중 하나를 살펴보자. 다음은 `<statement>` 기호를 파싱하는 방법을 보여준다.

```
parse_statement(tokens):
    expect("return", tokens)
    return_val = parse_exp(tokens)
    expect(";", tokens)
    return Return(return_val)

expect(expected, tokens):
    actual = take_token(tokens)
    if actual != expected:
        fail("Syntax error")
```

- 남은 토큰 목록이 `<statement>`로 시작할 것으로 예상되면, `parse_statement` 함수를 호출한다.
- 앞에서 봤던 형식 문법에 따르면, `<statement>`는 3가지 기호로 구성되어 있다.
  - `return` 키워드
  - `<exp>` 기호
  - `;` 토큰

# Recursive Descent Parsing

Korea University MatKor 스터디  
A Minimal Compiler

- 파싱 함수 중 하나를 살펴보자. 다음은 <statement> 기호를 파싱하는 방법을 보여준다.

```
parse_statement(tokens):
    expect("return", tokens)
    return_val = parse_exp(tokens)
    expect(";", tokens)
    return Return(return_val)

expect(expected, tokens):
    actual = take_token(tokens)
    if actual != expected:
        fail("Syntax error")
```

- 먼저 첫 번째 토큰이 실제로 return 키워드인지 확인하기 위해 헬퍼 함수 expect를 호출한다.
  - return 키워드라면 expect는 이를 제거하고 다음 토큰으로 진행할 수 있게 한다.
  - return 키워드가 아니라면 프로그램의 구문 오류를 보고한다.

# Recursive Descent Parsing

Korea University MatKor 스터디  
A Minimal Compiler

- 파싱 함수 중 하나를 살펴보자. 다음은 `<statement>` 기호를 파싱하는 방법을 보여준다.

```
parse_statement(tokens):
    expect("return", tokens)
    return_val = parse_exp(tokens)
    expect(";", tokens)
    return Return(return_val)

expect(expected, tokens):
    actual = take_token(tokens)
    if actual != expected:
        fail("Syntax error")
```

- 다음으로 `<exp>` 기호를 AST 노드 `exp`로 변환해야 한다.
  - `<exp>` 기호는 비종단 기호이므로 별도의 함수 `parse_exp`로 처리해야 한다. (이 함수는 여기서 정의하지 않았다.)
  - `parse_exp`를 호출하여 반환값으로 AST 노드를 얻는다.
- 그런 다음 `expect`를 다시 호출해 이 표현식 뒤에 마지막 토큰이 세미콜론인지 확인한다.
- 마지막으로 AST 노드 `Return`을 생성해 반환한다.

# Recursive Descent Parsing

Korea University MatKor 스터디  
A Minimal Compiler

- 파싱 함수 중 하나를 살펴보자. 다음은 <statement> 기호를 파싱하는 방법을 보여준다.

```
parse_statement(tokens):
    expect("return", tokens)
    return_val = parse_exp(tokens)
    expect(";", tokens)
    return Return(return_val)

expect(expected, tokens):
    actual = take_token(tokens)
    if actual != expected:
        fail("Syntax error")
```

- `parse_statement`는 문장을 구성하는 모든 토큰을 `tokens` 목록에서 제거한다.  
`parse_statement`가 반환된 후 호출자는 `tokens`에 남아 있는 토큰들을 계속 처리한다.
- 만약 프로그램 전체를 파싱한 후에도 토큰이 남아 있다면, 이는 구문 오류다.

# The Parser

Korea University MatKor 스터디  
A Minimal Compiler

- 파서 구현 관련 몇 가지 팁
  - 출력 형식을 알아보기 쉽게 작성하라.

```
Program(  
  Function(  
    name = "main",  
    body = Return(  
      Constant(2)  
    )  
  )  
)
```

- 도움이 되는 오류 메시지를 제공하라.
  - Fail vs Expected ";" but found "return"

# Assembly Generation

Korea University MatKor 스터디  
A Minimal Compiler

- AST를 x64 어셈블리로 변환해야 하며, 프로그램 실행 순서와 유사한 순서로 AST를 탐색해 각 노드에 대해 적절한 어셈블리 명령을 생성해야 한다.
  - 먼저, 파서를 작성할 때 AST를 표현하기 위해 자료 구조를 정의했던 것처럼 어셈블리 프로그램을 표현할 적절한 자료 구조를 정의한다.
  - 생성된 어셈블리 코드를 수정할 수 있도록, 어셈블리를 바로 파일에 작성하는 대신 또 다른 자료 구조를 추가한다.
- ASDL을 통해 어셈블리를 표현하기 위해 사용할 구조를 설명한다.

```
program = Program(function_definition)
function_definition = Function(identifier name, instruction* instructions)
instruction = Mov(operand src, operand dst) | Ret
operand = Imm(int) | Register
```

# Assembly Generation

Korea University MatKor 스터디  
A Minimal Compiler

- 다음은 각 AST 노드에 대해 생성해야 할 어셈블리 코드를 보여준다.

AST node	Assembly construct
Program(function_definition)	Program(function_definition)
Function(name, body)	Function(name, instructions)
Return(exp)	Mov(exp, Register) Ret
Constant(int)	Imm(int)

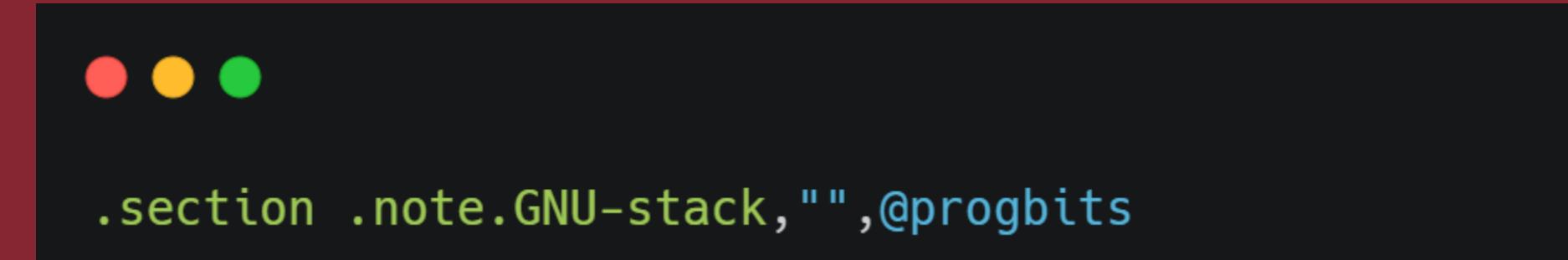
# Assembly Generation

---

Korea University MatKor 스터디  
A Minimal Compiler

- 주의할 점
  - 단일 문장이 여러 어셈블리 명령어로 변환된다.
  - 이 변환은 표현식이 단일 어셈블리 피연산자로 표현될 수 있을 때만 유효하다.
  - 현재는 표현식이 정수 상수뿐이므로 문제가 없지만, Lecture 2에서 단항 연산자를 추가하면 상황이 달라진다.  
그때부터는 컴파일러가 계산하기 위해 여러 명령어를 생성한 후,  
해당 표현식이 저장된 위치를 파악해 EAX로 복사해야 한다.

- 이제 컴파일러가 어셈블리 명령어를 실행할 수 있게 되었으니,  
마지막 단계는 해당 명령어를 파일로 작성하는 것이다.
  - 이 파일은 6번 슬라이드에서 봤던 어셈블리 프로그램과 매우 유사하지만, 몇 가지 세부 사항은 플랫폼에 따라 다르다.
  - 첫째, macOS 환경에서는 함수 이름 앞에 항상 밑줄(\_)을 추가해야 한다. (예를 들어, main 함수를 \_main으로 출력)
  - 둘째, Linux 환경에서는 파일 끝에 다음 줄을 추가해야 한다.



이 줄은 중요한 보안 강화 조치를 활성화해 코드에 실행 가능한 스택(Executable Stack)이 필요하지 않음을 나타낸다.

- 실행 가능하다(Executable)는 건 프로세서가 해당 메모리 영역에 저장된 기계 명령어의 실행을 허용했다는 걸 의미한다.
- 스택(Stack)은 지역 변수와 임시 값을 보관하는 메모리 영역이다. (Lecture 2에서 자세히 알아볼 예정)  
일반적인 상황에서는 기계 명령어를 저장하지 않는다.
- 스택을 실행 불가능하게 만드는 건 특정 보안 취약점에 대한 기본적인 방어 수단이다.

# Code Emission

- 코드 생성 단계는 어셈블리 AST를 순회하며 발견하는 각 구문을 출력해야 한다.  
(어셈블리 생성 단계에서 AST를 순회하는 방식과 유사하다.)
- 각 어셈블리 구문을 출력하는 방법은 다음과 같다.

Assembly top-level construct	Output
Program(function_definition)	Print out the function definition. On Linux, add at end of file: .section .note.GNU-stack,"",@progbits
Function(name, instructions)	.globl <name> <name>: <instructions>

Assembly instruction	Output
Mov(src, dst)	movl <src>, <dst>
Ret	ret

Assembly operand	Output
Register	%eax
Imm(int)	\$<int>

# Code Emission

---

Korea University MatKor 스터디  
A Minimal Compiler

- 코드 생성 구현 관련 몇 가지 팁
  - 명령어 사이에 줄바꿈을 반드시 포함하라.
  - 가독성이 좋고 잘 정렬된 어셈블리 코드를 생성하라.  
(컴파일러를 디버깅할 때 이 어셈블리 코드를 읽는데 많은 시간을 할애하게 되기 때문)
  - 어셈블리 프로그램에 주석을 포함하는 것도 고려하라.

# Assignment #1

- 다음 코드를 위한 Lexer, Parser, Assembly Generation, Code Emission 만들기



```
● ● ●

int main(void) {
    return 2;
}
```

- 각 단계에 해당하는 프로그램을 작성한 뒤 테스트 코드 실행을 통해 통과 여부를 확인할 수 있다.
  - Lexer : ./test\_compiler /path/to/your\_compiler --chapter 1 --stage lex
  - Parser : ./test\_compiler /path/to/your\_compiler --chapter 1 --stage parse
  - Assembly Generation : ./test\_compiler /path/to/your\_compiler --chapter 1 --stage codegen
  - 전체 : ./test\_compiler /path/to/your\_compiler --chapter 1

# 감사합니다.

utilForever@gmail.com

<https://github.com/utilForever>

X, Instagram: @utilForever