[KISTI 기술문서] ISBN 978-89-294-1717-8

LLVM 튜토리얼

2024. 10. 1.

한국과학기술정보연구원
슈퍼컴퓨팅기술개발센터

저자소개

김상완

한국과학기술정보연구원
국가슈퍼컴퓨팅본부 슈퍼컴퓨팅기술개발센터
책임연구원
sangwan@kisti.re.kr

정기문

한국과학기술정보연구원
국가슈퍼컴퓨팅본부 슈퍼컴퓨팅기술개발센터
책임연구원
kmjeong@kisti.re.kr

이 기술보고서는 2024년도 한국과학기술정보연구원(KISTI)의
기본사업으로 수행된 연구입니다.
과제번호: (KISTI) K24L2M1C6
과제명: 초고성능컴퓨팅 공동활용을 위한 통합 환경 개발 및 구축

목 차

1. 개요

- LLVM[1][2]은 컴파일러의 기반구조임. LLVM을 활용하여 어떤 프로그래밍 언어의 프론트엔드와 모든 명령어 세트 아키텍처의 백엔드를 개발할 수 있다. 언어 독립적인 중간 표현(IR, intermediate representation)을 중심으로 설계됨. 원래는 Low Level Virtual Machine을 의미했지만 프로젝트가 확장되어 더이상 공식적인 약어는 아님. LLVM은 C++로 작성되었으며 컴파일, 링크, 실행 시간에 걸쳐 최적화를 수행할 수 있는 유연한 컴파일러 기반구조를 제공하는 것을 목적으로 개발됨
- LLVM의 개발은 2000년 일리노이 대학교 어바나 캠퍼스에서 Vikram Adve[3]와 Chris Lattner의 지휘하에 시작됨. 2005년 Apple은 Lattner를 고용하여 LLVM을 기반으로 새로은 개발 도구를 개발할 팀을 구성. Lattner는 Clang 프로젝트를 시작하고 Objective-C 개발 책임을 맡음. 2007년 Mac OS X 10.5 Leopard 의 OpenGL 스택에서 JIT(Just-In-Time)[4] 컴파일러로 LLVM 기반 기술을 처음으로 제공. 2010년 부터 개발이 시작되어 2014년 6월 WWDC 2014에서 소개된 Swift 프로그래밍 언어도 LLVM 컴파일러를 기반으로 개발됨.
- 깃허브에서 LLVM 프로젝트 소스코드 및 튜토리얼 소스코드 가져오기.

```
$ git clone  https://github.com/llvm/llvm-project.git
$ cd  llvm-project/llvm/examples
```

- OCaml[5]은 강력하고 유연한 프로그래밍 언어로 LLVM에서 사용하고 있다. OCaml은 1996년에 Xavier Leroy 등이 개발한 함수형 프로그래밍 언어로 ML (Meta language) 언어 계열의 Caml 언어에 객체지향 기능을 추가한 것. 현재는 INRIA(프랑스 국립 자동화 연구소)에서 관리하는 오픈 소프 프로젝트임.
- OCaml의 강력한 타입 시스템과 함수형 프로그래밍 특성이 컴파일러 개발에 적합함.

- 본 문서에서 사용된 예제코드는 다음 깃허브 주소에서 찾을 수 있다:

```
git clone https://github.com/swkim85/linux-drill
cd linux-drill/llvm
```

1) https://www.llvm.org/
2) https://en.wikipedia.org/wiki/LLVM
3) https://en.wikipedia.org/wiki/Vikram_Adve
4) https://en.wikipedia.org/wiki/Just-in-time_compilation
5) https://ocaml.org/

2. Clang

- Clang(C language family frontend for LLVM)은 GCC[6]의 한계를 극복하기 위해서 개발이 시작됨. GCC는 오래동안 사용되어 왔지만, 코드베이스가 복잡하고 확장이 어려움. LLVM은 재사용 가능한 컴파일러 인프라를 목표 하기에 Clang은 모듈화된 설계로 더 쉽게 확장하고 유지보수 할 수 있도록 개발됨. Clang은 GCC 보다 더 유용하고 자세한 에러 및 경고 메시지를 제공하며 GCC 보다 빠른 컴파일 속도를 목표로 함. GCC의 GPL 라이선스와 달리 Clang은 더 유연한 BSD 라이선스를 사용함.
- 컴파일러는 프론트엔드와 백엔드로 나뉨. clang은 언어에 따라 컴파일해 주는 프론트엔드 부분을 담당. C/C++ 소스 코드는 clang을 통해 LLVM 비트코드(bitcode)로 컴파일됨
- llvm 과 clang 설치 방법

```
  우분투 24.04
$ sudo apt install  llvm  clang
$ dpkg-query -L clang
$ dpkg-query -L llvm
  버전 확인하기
$ llvm-config --version
18.1.3
$ clang --version
Ubuntu clang version 18.1.3 (1ubuntu1)
Target: x86_64-pc-linux-gnu
Thread model: posix
InstalledDir: /usr/bin
```

2.1 비트코드

- Clang 테스트 예제 프로그램

```
// helloworld.c
#include <stdio.h>
int add(int a, int b) {
  int sum;
  sum = a + b;
  return sum;
}
int main() {
  int number = add(123, 456);
  printf("Hello, number %d!\n", number);
  return 0;
}
$ clang  helloworld.c  -o helloworld            실행파일 만들기(컴파일 + 링크)
$ ./helloworld
Hello, world!

  소스코드를 bitcode 형태로 바꾸기 (-emit-llvm)
$ clang -emit-llvm  -c helloworld.c -o helloworld.bc
$ file helloworld.bc
helloworld.bc: LLVM IR bitcode

 소스코드를 사람이 읽을 수 있는 IR코드로 바꿈 (-emit-llvm -S)
$ clang -emit-llvm -S helloworld.c  -o helloworld.ll
$ more helloworld.ll
```

6) https://gcc.gnu.org/

```
; ModuleID = 'helloworld.c'
source_filename = "helloworld.c"
  target datalayout [7]은  데이터의 메모리 레이아웃을 지정한다. e는 little endian을 의미. "m:e"는 mangling 으로 e는 ELF 방식으로 개인 심볼 뒤에 .L접두사가 붙음. p는 주소공간에서 포인터의 크기와 정렬을 나타낸다. i는 정수 타입의 크기, f는 실수 타입의 크기, n는 native integer의 크기, S는 스택 정렬 크기임.
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-i128:128-f80:128-n8:16:32:64-S128"
  타켓 트리플[8]은 타겟 호스트의 정보로 형식은 ARCHITECTURE-VENDOR-OPERATING_SYSTEM-ENVIRONMENT
target triple = "x86_64-pc-linux-gnu"

 @는 전역 변수, .str은 문자열을 의미. private는 현재 모듈 내에서만 접근 가능함. unnamed_addr는 주소 값이 중요하지 않음. constant 변경되지 않는 상수임. [19xi8] 은 19개의 8비트(char) 배열임을 의미.
@.str = private unnamed_addr constant [19 x i8] c"Hello, number %d!\0A\00", align 1

  define [9] 은 함수를 정의함
  dso_local로 표시된 함수나 변수가 동일한 연결 단위 내의 심볼임을 의미
  i32 는 정수 타입 [10]
  함수명 앞에 @는 전역 식별자임을 나타냄. 지역 식별자는 %로 시작.
  noundef 는 함수 매개변수나 반환값에 대한 속성으로 undefined가 아님을 의미 [11]
  함수 정의 맨 뒤의 #0는 속성 그룹으로 [12] 함수에 적용되는 속성들의 집합을 참조한다. 동일한 속성의 집합을 여러 함수에 적용하여 재사용하고, IR의 크기를 줄이고 가독성을 향상시키기 위함.
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @add(i32 noundef %0, i32 noundef %1) #0 {
  alloca 명령은 스택 메모리를 동적으로 할당하는데 사용됨. 로컬 변수를 위한 메모리 공간 확보 [13]
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
    store 명령은 특정 메모리 주소에 값을 저장하는 역할 [14]
    ptr 은 포인터 타입을 나타냄. LLVM 14버전부터 도입된 표기법으로 이전 버전에서는 "i32*"와 같이 표현했었음
  store i32 %0, ptr %3, align 4
  store i32 %1, ptr %4, align 4
    load 명령은 지정된 메모리 주소에서 값을 읽어 레지스터나 변수에 저장 [15]
  %6 = load i32, ptr %3, align 4
  %7 = load i32, ptr %4, align 4
   nsw 는 no signed wrap의 약자로 부호 있는 정수 오버플로우를 무시하는 플래그임 [16]
  %8 = add nsw i32 %6, %7
  store i32 %8, ptr %5, align 4
  %9 = load i32, ptr %5, align 4
    ret [17]는 control flow를 함수에서 caller 에게 되돌려 준다.
  ret i32 %9
}

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  store i32 0, ptr %1, align 4
    call 명령: 함수 호출의 결과를 저장함. [18]
  %3 = call i32 @add(i32 noundef 123, i32 noundef 456)
  store i32 %3, ptr %2, align 4
  %4 = load i32, ptr %2, align 4
```

```
  %5 = call i32 (ptr, ...) @printf(ptr noundef @.str, i32 noundef %4)
  ret i32 0
}
  declare 키워드는 함수 선언을 나타냄. 외부에서 정의된 함수의 인터페이스를 선언함.
declare i32 @printf(ptr noundef, ...) #1

 함수의 속성 그룹 (정의)
attributes #0 = { noinline nounwind optnone uwtable "frame-pointer"="all" "min-legal-vector-wid
th"="0" "no-trapping-math"="true" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "targ
et-features"="+cmov,+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }
  frame-pointer 속성은 스택 프레임 포인터의 사용 여부를 제어
  no-trapping-math 속성은 부동소수점 연산의 최적화 연산의 처리 방식을 결정
  stack-protector-buffer-size 속성은 스택 보호 매커니즘과 관련.
attributes #1 = { "frame-pointer"="all" "no-trapping-math"="true" "stack-protector-buffer-size"
="8" "target-cpu"="x86-64" "target-features"="+
cmov,+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "tune-cpu"="generic" }

!llvm.module.flags = !{!0, !1, !2, !3, !4}
!llvm.ident = !{!5}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{i32 8, !"PIC Level", i32 2}    Position Independent Code (PIC) 레벨을 2로 설정
!2 = !{i32 7, !"PIE Level", i32 2}    Position Independent Executable (PIE) 레벨을 2로 설정
!3 = !{i32 7, !"uwtable", i32 2}      Unwind Table 생성을 활성화
!4 = !{i32 7, !"frame-pointer", i32 2}   프레임 포인터 사용을 설정
!5 = !{!"Ubuntu clang version 18.1.3 (1ubuntu1)"}
```

```
 .ll 파일을 컴파일하여 바이너리 만들기
$ clang  helloworld.ll  -o helloworld2
$ diff helloworld  helloworld2     두 바이너리 파일은 동일함
$ file helloworld              바이너리 파일의 file 정보
helloworld: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interp
reter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=7a8ff1d062e662e324d9a5357fc73c6c12c6c102, for
GNU/Linux 3.2.0, not stripped
```

- llvm-as(어셈블러), llvm-dis(역어셈블러)를 사용하여 LLVM IR의 형태를 변환할 수 있다

```
  .ll 에서 .bc 로 변환 (어셈블링 텍스트 -> 바이너리)
$ llvm-as  helloworld.ll  -o helloworld.bc
  .bc 에서 .ll 로 변환 (역어셈블링 바이너리 -> 텍스트)
$ llvm-dis  helloworld.bc  -o helloworld.ll
```

- llc를 사용하여 IR을 타겟 머신의 바이너리(어셈블리)로 컴파일(compile+link) 할 수 있다

7) https://llvm.org/docs/LangRef.html#langref-datalayout
8) https://llvm.org/docs/LangRef.html#target-triple
9) https://llvm.org/docs/LangRef.html#functions
10) https://llvm.org/docs/LangRef.html#integer-type
11) https://llvm.org/docs/LangRef.html#parameter-attributes
12) https://llvm.org/docs/LangRef.html#attribute-groups
13) https://llvm.org/docs/LangRef.html#alloca-instruction
14) https://llvm.org/docs/LangRef.html#store-instruction
15) https://llvm.org/docs/LangRef.html#load-instruction
16) https://llvm.org/docs/LangRef.html#add-instruction
17) https://llvm.org/docs/LangRef.html#ret-instruction
18) https://llvm.org/docs/LangRef.html#call-instruction

```
$ llc  -mtriple=x86_64-pc-linux-gnu   helloworld.ll
$ more helloworld.s
        .text
        .file   "helloworld.c"
        .globl  add                             # -- Begin function add
[...]
```

※ ARM 플랫폼으로 크로스컴파일 하려면. 리눅스용 ARM 툴체인을 설치하고 clang 에서 -target 옵션을 지정하면 됨

```
   ARM 툴체인을 설치한다.
$ sudo apt-get install gcc-arm-linux-gnueabihf
   clange 에서 target 을 지정하여 비트코드를 생성
$ clang -target arm-linux-gnueabihf -S -emit-llvm helloworld.c -o helloworld.ll
  IR 코드를 ARM 어셈블리로 변환
$ llc -march=arm -mcpu=cortex-a9 helloworld.ll -o helloworld.s
  ARM 어셈블리를 실행파일로 링크
$ arm-linux-gnueabihf-gcc helloworld.s -o helloworld
```

※ GCC와 Clang의 차이: 초기화되지 않은 변수에 대한 경고

```
// ex1.c
#include <stdio.h>
int main() {
    int x;    // 변수가 초기화 되지 않음
    printf("%d\n", x);   return 0;
}
```

```
$ gcc -Wall ex1.c
ex1.c: In function ‘main’:
ex1.c:6:5: warning: ‘x’ is used uninitialized [-Wuninitialized]
    6 |     printf("%d\n", x);
      |     ^~~~~~~~~~~~~~~~~
ex1.c:5:9: note: ‘x’ was declared here
    5 |     int x;
      |         ^
```

```
$ clang -Wall ex1.c           경고 메시지의 내용이 GCC와 다름을 알 수 있다
ex1.c:6:20: warning: variable 'x' is uninitialized when used here [-Wuninitialized]
    6 |     printf("%d\n", x);
      |                    ^
ex1.c:5:10: note: initialize the variable 'x' to silence this warning
    5 |     int x;
      |          ^
      |           = 0
1 warning generated.
```

※ GCC와 Clang의 차이: 최적화 수준 차이

```
// ex4.c
int foo (int num) {       // num이 홀수이면 제곱하고, 짝수이면 제곱에 1을 더함
    if (num % 2 == 1)   return num * num;
    else      return num * num +1;
}
```

```
   최적화 -O2 옵션으로 각각 gcc 와 clang 으로 어셈블리로 변환
$ gcc   -O2 -S ex4.c -o ex4.s.gcc13
$ clang -O2 -S ex4.c -o ex4.s.clang18
```

```
$ more  ex4.s.gcc13
[...]
foo:
.LFB0:
   .cfi_startproc
   endbr64
   movl    %edi, %edx   edi를 edx로 복사
   movl    %edi, %eax   edi를 eax로 복사
   shrl    $31, %edx   edx를 31비트 오른쪽으로
쉬프트. 원래 값의 부호비트를 추출
   imull   %edi, %eax   제곱한 값을 eax에 저장
   addl    %edx, %edi   부호비트를 edi에 더함
   andl    $1, %edi     edi 최하위 비트만 남김
   subl    %edx, %edi   edi에서 부호비트를 빼기
   xorl    %edx, %edx  edx를 0으로 초기화
   cmpl    $1, %edi    edi와 1을 비교
   setne   %dl       이전 비교 결과가 같지 않으
면 dl 을 1로 설정, 같으면 0으로 설정
   addl    %edx, %eax  edx를 eax에 더함
   ret               함수에서 반환
   .cfi_endproc  [...]
```

```
$ more ex4.s.clang18
[...]
foo:                                    # @foo
    .cfi_startproc
# %bb.0:
    movl    %edi, %ecx    edi를 ecx로 복사
    andl    $-2147483647, %ecx
            # imm = 0x80000001
       최상위 비트와 최하위 비트만 유지
    xorl    %eax, %eax
    cmpl    $1, %ecx     ecx를 1과비교
    setne   %al          이전 비교결과가 같지
않으면 AL을 1로 설정, 같으면 0으로 설정
    imull   %edi, %edi    edi를 자신과 곱하여 e
di에 저장 (제곱연산)
    addl    %edi, %eax    edi값에 eax를 더함
    retq                  eax 값을 반환

[...]
```

※ llvm-bcanalyzer 툴은 .bc파일을 분석하여 통계적인 레포트를 출력한다.

```
$ llvm-bcanalyzer --help                  도움말
OVERVIEW: llvm-bcanalyzer file analyzer

USAGE: llvm-bcanalyzer [options] <input bitcode>
[...]
$ llvm-bcanalyzer  [--dump]  helloworld.bc
Summary of helloworld.bc:
         Total size: 22048b/2756.00B/689W
        Stream type: LLVM IR
  # Toplevel Blocks: 4

Per-block Summary:
  Block ID #0 (BLOCKINFO_BLOCK):
      Num Instances: 1
         Total Size: 800b/100.00B/25W
    Percent of file: 3.6284%
      Num SubBlocks: 0
        Num Abbrevs: 19
        Num Records: 3
    Percent Abbrevs: 0.0000%
[...]
```

3. LLVM 튜토리얼

■ 튜토리얼의 목적

- LLVM 튜토리얼[19]에서는 Kaleidoscope(만화경 이라는 뜻. 이하 Kal)[20]라는 실험용 언어(toy language)를 기반으로 작성되었다. Kal은 절차형 언어(procedural language)로 함수를 정의하고, 조건문과 수학 연산들을 이용할 수 있다. 튜토리얼에서는 Kal을 확장하여 if/then/else 구조, for/loop 반복문, 사용자 정의 연산자, JIT 컴파일과 디버깅 정보등을 지원할 수 있도록 한다. 문제를 단순화하기 위해서 Kal에서 사용하는 데이터 타입은 오직 64비트 실수형 포인트만 지원하기로 한다.
- 아래 코드는 Kal 언어를 이용하여 피보나치수[21]를 계산하는 코드이다.

```
# 피보나치 수열(fibonacci sequence)에서 x번째의 값을 구함 fib[1]=1, fib[2]=1, fib[3]=2,...
def fib(x)
  if x < 3 then
    1
  else
    fib(x-1)+fib(x-2)
# This expression will compute the 40th number.
fib(40)
```

- 또한 Kal 언어는 LLVM JIT(just-in-time, 즉시 컴파일 또는 즉시 실행)을 이용하여 표준 라이브러리 함수를 호출할 수 있다. extern 키워드를 이용하여 함수 선언을 먼저 표기한다.

```
extern sin(arg);
extern cos(arg);
atan2(sin(.4), cos(42))
```

3.1. 어휘분석기

- Lexer(렉서, 어휘분석기)는 컴파일러나 인터프리터에서 소스 코드를 처리하는 초기 단계에서 사용되는 컴포넌트로, lexical analysis(어휘 분석)를 수행. 소스 코드를 읽어 들여 토큰(token)이라는 기본 단위로 나누는 역할을 함. 이 과정은 다음 단계인 파서(parser)가 구문 분석을 수행하기 위해 필수적인 전처리 단계이다.

```
// 알려지지 않은 문자의 경우 토큰 [0-255](아스키코드)를 리턴
enum Token {
  tok_eof = -1,
  // commands
  tok_def = -2,  tok_extern = -3,
  // primary
  tok_identifier = -4,  tok_number = -5,
};
static std::string IdentifierStr; // Filled in if tok_identifier
static double NumVal;             // Filled in if tok_number
```

- 렉서의 실제 구현은 gettok() 이라는 단일 함수임.

```
/// gettok - 표준 입력에서 다음 토큰을 받아서 리턴 (Tokenizer)
static int gettok() {
```

19) https://llvm.org/docs/tutorial/
20) https://en.wikipedia.org/wiki/Kaleidoscope_(programming_language)
21) https://en.wikipedia.org/wiki/Fibonacci_number

```
  static int LastChar = ' ';
  // Skip any whitespace.
  while (isspace(LastChar))    LastChar = getchar();

  if (isalpha(LastChar)) { // identifier: [a-zA-Z][a-zA-Z0-9]*
    IdentifierStr = LastChar;
    while (isalnum((LastChar = getchar())))
      IdentifierStr += LastChar;

    if (IdentifierStr == "def") // 토큰 def
      return tok_def;
    if (IdentifierStr == "extern") // 토큰 extern
      return tok_extern;
    return tok_identifier;
  }

  if (isdigit(LastChar) || LastChar == '.') { // 숫자 [0-9.]+
    std::string NumStr;
    do {
      NumStr += LastChar;
      LastChar = getchar();
    } while (isdigit(LastChar) || LastChar == '.');
    NumVal = strtod(NumStr.c_str(), nullptr);  // 값은 NumVal 이라는 변수에 저장
    return tok_number;
  }

  if (LastChar == '#') {  // 코멘트. 라인의 끝까지
    do
      LastChar = getchar();
    while (LastChar != EOF && LastChar != '\n' && LastChar != '\r');
    if (LastChar != EOF)      return gettok();
  }

  if (LastChar == EOF)  // end of file 인지 체크. EOF를 삼키지는 않음
    return tok_eof;

  // Otherwise, just return the character as its ascii value.
  int ThisChar = LastChar;
  LastChar = getchar();
  return ThisChar;
}
```

- 렉서를 컴파일하고 테스트하기

```
$ clang++ -g -O3 tut1.cpp  -o tut1
$ ./tut1
ready> 1+2;
[tut1]CurTok : -5         <------ 숫자 1은 tok_number(-5)
ready> [tut1]CurTok : 43  <--- 더하기(+) 은 아스키 코드 43임
[tut1]CurTok : -5    <------ 숫자 2은 tok_number(-5)
[tut1]CurTok : 59   <--- 세미콜론(;) 은 아스키 코드 59임
Parsed a top-level expr
ready> 1;
[tut1]CurTok : -5     <------ 숫자 1은 tok_number(-5)
ready> [tut1]CurTok : 59
```

```
Parsed a top-level expr
ready> extern a(b);
[tut1]CurTok : -3    <------ 토큰 "extern" tok_extern(-3)
ready> [tut1]CurTok : -4   <---- 식별자 "a" tok_identifier(-4)
[tut1]CurTok : 40    <----- 여는괄호 "("
[tut1]CurTok : -4  <---- 식별자 "b" tok_identifier(-4)
[tut1]IdentifierStr : b
[tut1]CurTok : 41  <----- 닫는괄호 ")"
[tut1]CurTok : 59  <--- 세미콜론(;)
Parsed an extern
ready> ^C
```

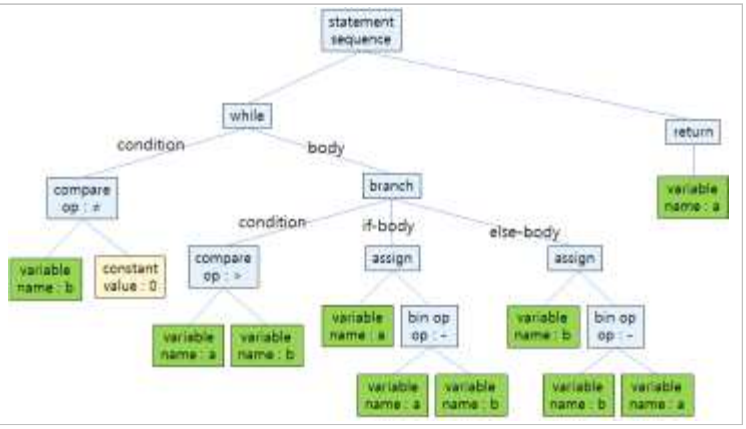
3.2. 파서와 AST

- AST(Abstract Syntax Tree, 추상구문트리)는 프로그래밍 언어의 소스 코드를 구조적으로 표현하는 트리 형태의 데이터 구조임.[22] AST 생성을 위한 과정: ①파싱: 소스 코드를 토큰으로 분리한다. ②구문 분석: 토큰을 기반으로 구문 구조를 파악 ③AST 구축: 분석된 구조를 바탕으로 트리를 생성

예시) Euclidean 알고리즘

```
while b ≠ 0:
    if a > b:
        a := a - b
    else:
        b := b - a
return a
```

위 코드를 AST로 분석하면 오른쪽 그림과 같다:



※ AST를 시각적으로 보는 방법. `-Xclang -ast-dump` 옵션을 이용하면 프로그램의 AST를 볼 수 있다. `helloworld.c` 코드의 구문트리를 출력하기.

```
$ clang -Xclang -ast-dump  -fno-color-diagnostics  -fsyntax-only  helloworld.c
[...]
`-FunctionDecl 0x58ccf39b8988 <line:10:1, line:14:1> line:10:5 main 'int ()'
  `-CompoundStmt 0x58ccf39b8d60 <col:12, line:14:1>
    |-DeclStmt 0x58ccf39b8b88 <line:11:3, col:29>
    | `-VarDecl 0x58ccf39b8a48 <col:3, col:28> col:7 used number 'int' cinit
    |   `-CallExpr 0x58ccf39b8b58 <col:16, col:28> 'int'
    |     |-ImplicitCastExpr 0x58ccf39b8b40 <col:16> 'int (*)(int, int)' <FunctionToPointerDecay>
    |     | `-DeclRefExpr 0x58ccf39b8ab0 <col:16> 'int (int, int)' Function 0x58ccf39b8698 'add' 'int (int, int)'
    |     |-IntegerLiteral 0x58ccf39b8ad0 <col:20> 'int' 123
    |     `-IntegerLiteral 0x58ccf39b8af0 <col:25> 'int' 456
    |-CallExpr 0x58ccf39b8cb8 <line:12:3, col:39> 'int'
    | |-ImplicitCastExpr 0x58ccf39b8ca0 <col:3> 'int (*)(const char *, ...)' <FunctionToPointerDecay>
    | | `-DeclRefExpr 0x58ccf39b8ba0 <col:3> 'int (const char *, ...)' Function 0x58ccf39791a8 'printf' 'int (const cha
r *, ...)'
| |-ImplicitCastExpr 0x58ccf39b8d00 <col:10> 'const char *' <NoOp>
    | | `-ImplicitCastExpr 0x58ccf39b8ce8 <col:10> 'char *' <ArrayToPointerDecay>
    | |   `-StringLiteral 0x58ccf39b8c00 <col:10> 'char[19]' lvalue "Hello, number %d!\n"
    | `-ImplicitCastExpr 0x58ccf39b8d18 <col:33> 'int' <LValueToRValue>
    |   `-DeclRefExpr 0x58ccf39b8c30 <col:33> 'int' lvalue Var 0x58ccf39b8a48 'number' 'int'
```

22) https://en.wikipedia.org/wiki/Abstract_syntax_tree

```
    `-ReturnStmt 0x58ccf39b8d50 <line:13:3, col:10>
      `-IntegerLiteral 0x58ccf39b8d30 <col:10> 'int' 0
```

- 재귀적 하강 파싱(recursive descent parsing)[23]과 연산자 우선 순위 파싱(operator-precedence parsing)[24]의 조합을 사용하여 Kal언어를 파싱한다. AST에서는 언어의 각 구성 요소에 대해 하나의 객체를 원하며, 따라서 AST는 언어를 면밀히 모델링 해야 함. Kal언어에는 표현식, 프로토타입, 함수 객체가 있음.
- 표현식(expression)을 위한 AST 객체를 정의

```
/// ExprAST - 모든 표현식 클래스의 베이스 클래스
class ExprAST {
public:
  virtual ~ExprAST() = default;
};
/// 숫자 표현식 - Expression class for numeric literals like "1.0".
class NumberExprAST : public ExprAST {
  double Val;  // 숫자의 실제 값
public:
  NumberExprAST(double Val) : Val(Val) {}
};
```

- 변수(variable) 표현식, 이진 연산자(binary operator) 표현식, 함수호출 표현식 클래스

```
/// 변수 표현식 - Expression class for referencing a variable, like "a".
class VariableExprAST : public ExprAST {
  std::string Name;  // 변수의 이름
public:
  VariableExprAST(const std::string &Name) : Name(Name) {}
};
/// 이진 연산자 표현식 - Expression class for a binary operator.
class BinaryExprAST : public ExprAST {
  char Op;   // 연산자
  std::unique_ptr<ExprAST> LHS, RHS; // 좌변, 우변
public:
  BinaryExprAST(char Op, std::unique_ptr<ExprAST> LHS,
                         std::unique_ptr<ExprAST> RHS)
    : Op(Op), LHS(std::move(LHS)), RHS(std::move(RHS)) {}
};
/// 함수 호출 표현식 - Expression class for function calls.
class CallExprAST : public ExprAST {
  std::string Callee;   // 호출할 함수명
  std::vector<std::unique_ptr<ExprAST>> Args;  // 인수 목록 벡터, 인수 또한 표현식임.
public:
  CallExprAST(const std::string &Callee, std::vector<std::unique_ptr<ExprAST>> Args)
    : Callee(Callee), Args(std::move(Args)) {}
};
```

- 함수 프로토타입 표현식.

```
/// PrototypeAST - 어떤 함수의 프로토타입. 이름과 매개변수의 이름(들)을 저장
class PrototypeAST {
  std::string Name;  // 함수명
```

23) https://en.wikipedia.org/wiki/Recursive_descent_parser
24) https://en.wikipedia.org/wiki/Operator-precedence_parser

```
  std::vector<std::string> Args;  // 인수 리스트. 문자열 벡터
public:
  PrototypeAST(const std::string &Name, std::vector<std::string> Args)
    : Name(Name), Args(std::move(Args)) {}
  const std::string &getName() const { return Name; }
};
/// FunctionAST - 함수 정의
class FunctionAST {
  std::unique_ptr<PrototypeAST> Proto; // 함수 프로토타입
  std::unique_ptr<ExprAST> Body;  // 함수의 정의 내용 (표현식)
public:
  FunctionAST(std::unique_ptr<PrototypeAST> Proto, std::unique_ptr<ExprAST> Body)
    : Proto(std::move(Proto)), Body(std::move(Body)) {}
};
```

- 파서 코드 정의. “x+y" (렉서에 의해 3개의 토큰으로 변환됨)와 같은 표현식은 다음과 같은 호출로 생성될 수 있는 AST로 표현될 수 있음

```
auto LHS = std::make_unique<VariableExprAST>("x");  // 이름이 "x"인 변수
auto RHS = std::make_unique<VariableExprAST>("y");  // 이름이 "y"인 변수
auto Result = std::make_unique<BinaryExprAST>('+', std::move(LHS), std::move(RHS));
   // 이진연산자 ‘+’(더하기), 좌항(LHS) 피연산자와 우항(RHS) 피연산자로 구성
```

- 토큰 버퍼 구현. 파서의 오류 처리를 위한 도우미 루틴. 파서의 모든 함수는 CurTok 값이 현재 파싱해야 할 토큰이라고 가정함.

```
/// CurTok/getNextToken - 가장 단순한 토큰 버퍼를 구현 CurTok은 파서가 위치한 현재의 토큰, getN
extToken은 렉서로부터 다음 토큰을 전달받아 CurTok을 대체한다.
static int CurTok;
static int getNextToken() {
  return CurTok = gettok();
}
```

- 오류 처리 루틴 LogError은 파서가 오류는 처리하는 데 사용됨.

```
/// LogError* - These are little helper functions for error handling.
std::unique_ptr<ExprAST> LogError(const char *Str) {  // ExprAST 타입을 리턴
  fprintf(stderr, "Error: %s\n", Str);
  return nullptr;  // 항상 null을 반환함
}
std::unique_ptr<PrototypeAST> LogErrorP(const char *Str) {   // PrototypeAST 타입을 리턴
  LogError(Str);
  return nullptr;  // 항상 null을 반환함
}
```

- 숫자 표현식 구문 분석

```
/// numberexpr ::= number
static std::unique_ptr<ExprAST> ParseNumberExpr() {
  auto Result = std::make_unique<NumberExprAST>(NumVal);  // NumVal은 전역 변수
  getNextToken(); // consume the number
  return std::move(Result);    // NumberExprAST 노드(표현식)를 만들어 리턴함
}
```

- 괄호 표현식. 닫는 괄호 ”)“를 체크하여 LogError를 발생시킨다.

```
/// parenexpr ::= '(' expression ')'
static std::unique_ptr<ExprAST> ParseParenExpr() {
  getNextToken(); // eat (.
  auto V = ParseExpression();  // 괄호안의 표현식을 파싱한다. 재귀적 호출로 연결된다.
  if (!V)    return nullptr;

  if (CurTok != ')')    return LogError("expected ')'");
  getNextToken(); // eat ).
  return V;  // 표현식을 리턴
}
```

- 변수 참조와 함수 호출을 처리

```
/// identifierexpr
///   ::= identifier
///   ::= identifier '(' expression* ')'
static std::unique_ptr<ExprAST> ParseIdentifierExpr() {
  std::string IdName = IdentifierStr;
  getNextToken();  // eat identifier.

  if (CurTok != '(')   // 괄호가 아니면 변수 참조
    return std::make_unique<VariableExprAST>(IdName); // 단순한 변수 => VariableExprAST

  // Call.  괄호 '('가 있으면 함수 호출로 간주
  getNextToken();  // eat '('
  std::vector<std::unique_ptr<ExprAST>> Args;
  if (CurTok != ')') {
    while (true) {
      if (auto Arg = ParseExpression())
        Args.push_back(std::move(Arg));  // 함수 호출 인자
      else        return nullptr;
      if (CurTok == ')')        break;  // 닫는 괄호 이므로 중지
      if (CurTok != ',')
        return LogError("Expected ')' or ',' in argument list");
      getNextToken();
    }
  }
  getNextToken();  // Eat the ')'.
  return std::make_unique<CallExprAST>(IdName, std::move(Args)); // 함수 호출 => CallExprAST
}
```

- 구문 분석의 진입점이 되는 도우미 함수를 정의. 이것을 "primary" 주표현식이라고 명명함.

```
/// primary
///   ::= identifierexpr
///   ::= numberexpr
///   ::= parenexpr
static std::unique_ptr<ExprAST> ParsePrimary() {
  switch (CurTok) {
  default:
    return LogError("unknown token when expecting an expression");
  case tok_identifier:
    return ParseIdentifierExpr();
  case tok_number:
    return ParseNumberExpr();
```

```
  case '(':
    return ParseParenExpr();
  }
}
```

- 이진 연산자의 경우는 위의 것들보다는 약간 더 복잡함. 이진 연산자 우선순위를 고려해야 하기 때문. 예를 들어 문자열 "x+y*z"가 주어졌을 때 파서는 "(x+y)*z" 또는 "x+(y*z)"로 파싱하도록 선택할 수 있음. 이를 위해 연산자 우선순위 파싱[25]을 사용해야 함.

```
/// BinopPrecedence - 이진 연산자의 우선순위를 위한 맵(key-value 연관 컨테이너)
static std::map<char, int> BinopPrecedence;

/// GetTokPrecedence - Get the precedence of the pending binary operator token.
static int GetTokPrecedence() {
  if (!isascii(CurTok))    return -1;
  // Make sure it's a declared binop.
  int TokPrec = BinopPrecedence[CurTok];
  if (TokPrec <= 0) return -1;
  return TokPrec;
}
int main() {
  // Install standard binary operators.
  // 1 is lowest precedence.
  BinopPrecedence['<'] = 10;
  BinopPrecedence['+'] = 20;  // +,- 보다는 * 가 더 높음 우선순위를 가짐
  BinopPrecedence['-'] = 20;
  BinopPrecedence['*'] = 40;  // highest.
  ...
}
```

- 주표현식(primary expression)뒤에는 [binop, primaryexpr]이 뒤 따라 올 수 있음. 예를 들어 "a+b+(c+d)*e*f+g" 라는 표현식에서 이진연산자의 앞과 뒤에는 주표현식이 위치한다. 파서는 처음에 "a"라는 주표현식을 보게되고, 이어서 [+,b]를 보게 된다. 이어서 [+,(c+d)] [*,e] [*,f] [+,g]를 순차적으로 보게 된다. 이들은 모두 [이진연산자, 주표현식]의 쌍으로 구성됨.

```
static std::unique_ptr<ExprAST> ParseExpression() {
  auto LHS = ParsePrimary();
  if (!LHS)    return nullptr;
  return ParseBinOpRHS(0, std::move(LHS));  // ExprPrec=0 으로 호출함에 주의
// ParseBinOpRHS를 호출할 때 연산자 우선순위와 지금까지 파싱한 주표현식에 대한 포인터를 전달함
}
/// binoprhs
///   ::= ('+' primary)*
static std::unique_ptr<ExprAST> ParseBinOpRHS (int ExprPrec, std::unique_ptr<ExprAST> LHS) {
  // If this is a binop, find its precedence.
  while (true) {
    int TokPrec = GetTokPrecedence();  // 이진 연산자의 우선순위를 반환
    // 현재 연산자의 우선순위가 더 크다면 여기서 끝낸다.
    if (TokPrec < ExprPrec)  return LHS;  // ----------- (1)

    // Okay, we know this is a binop.
    int BinOp = CurTok;
```

25) https://en.wikipedia.org/wiki/Operator-precedence_parser

```
    getNextToken(); // eat binop

    // 이진 연산자 오른쪽의 표현식을 파싱
    auto RHS = ParsePrimary();
    if (!RHS)  return nullptr;

    // If BinOp binds less tightly with RHS than the operator after RHS, let
    // the pending operator take RHS as its LHS.
    int NextPrec = GetTokPrecedence();
    if (TokPrec < NextPrec) {  // -------(2)
      RHS = ParseBinOpRHS(TokPrec+1, std::move(RHS));  // 재귀적 호출
      if (!RHS)
        return nullptr;
    }
    // LHS/RHS 둘을 합친다.
    LHS = std::make_unique<BinaryExprAST>(BinOp, std::move(LHS), std::move(RHS));
  }
}
```

```
$ clang++ -g -O3 tut2.cpp `llvm-config --cxxflags --ldflags --system-libs --libs core` -o tut2
$ ./tut2
ready> a+b*c;
[tut2]CurTok : -4
ready> [tut2]CurTok : 43   <---  ASCII 코드 43 은 +
[tut2]TokPrec: 20  <---- + 연산자 우선순위는 20
     ExprPrec=0  TokPrec=20   if (TokPrec < ExprPrec)=false이므로 (1) 이므로 계속진행
[tut2]CurTok : -4
[tut2]CurTok : 42  <---  ASCII 코드 42 는 *
[tut2]TokPrec: 40  <---- * 연산자 우선순위는 40
     TokPrec=20  NextPrec=40  if (TokPrec < NextPrec)=true 이므로 (2) 뒷부분(RHS)을 먼저 파싱한
다음  LHS와 묶는다
[tut2]CurTok : -4
[tut2]CurTok : 59  <---- 세미콜론
[tut2]TokPrec: -1
[tut2]TokPrec: -1
[...]
```

```
ready> a*b+c;
[tut2]CurTok : -4
ready> [tut2]CurTok : 42  <---  아스키 코드 42 는 *
[tut2]TokPrec: 40 <---- * 연산자 우선순위는 40
     ExprPrec=0  TokPrec=40   if (TokPrec < ExprPrec)=false이므로 (1) 이므로 계속진행
[tut2]CurTok : -4
[tut2]CurTok : 43  <---  아스키 코드 43 은 +
[tut2]TokPrec: 20  <---- + 연산자 우선순위는 20
 TokPrec=40  NextPrec=20 if (TokPrec < NextPrec)=false 이므로(2) +이후는 파싱하지 않고 a*b만 리턴
[tut2]CurTok : -4
[tut2]CurTok : 59
[tut2]TokPrec: -1
[...]
```

- 함수 프로토타입 처리. 'extern' 함수 선언과 함수 정의에서 사용된다.

```
/// prototype
///   ::= id '(' id* ')'
static std::unique_ptr<PrototypeAST> ParsePrototype() {
  if (CurTok != tok_identifier)
```

```
    return LogErrorP("Expected function name in prototype");

  std::string FnName = IdentifierStr;  // 함수명
  getNextToken();

  if (CurTok != '(')
    return LogErrorP("Expected '(' in prototype");

  // Read the list of argument names.
  std::vector<std::string> ArgNames;
  while (getNextToken() == tok_identifier)
    ArgNames.push_back(IdentifierStr);
  if (CurTok != ')')
    return LogErrorP("Expected ')' in prototype");

  // success.
  getNextToken();  // eat ')'.
  return std::make_unique<PrototypeAST>(FnName, std::move(ArgNames));
}
```

- 함수 정의 처리. 함수는 프로토타입과 본체(body)로 구성된다. 'extern' 함수 선언은 본체(body)가 없는 함수로 취급한다.

```
/// definition ::= 'def' prototype expression
static std::unique_ptr<FunctionAST> ParseDefinition() {
  getNextToken();  // eat def.
  auto Proto = ParsePrototype();
  if (!Proto) return nullptr;

  if (auto E = ParseExpression())  // 함수 정의는 하나의 표현식으로 간주
    return std::make_unique<FunctionAST>(std::move(Proto), std::move(E));
  return nullptr;
}
/// external ::= 'extern' prototype
static std::unique_ptr<PrototypeAST> ParseExtern() {
  getNextToken();  // eat extern.
  return ParsePrototype();
}
```

- 마지막으로 임의의 top-level 표현식을 정의한다.

```
/// toplevelexpr ::= expression
static std::unique_ptr<FunctionAST> ParseTopLevelExpr() {
  if (auto E = ParseExpression()) {
    // Make an anonymous proto. 이름이 없는 프로토타입
    auto Proto = std::make_unique<PrototypeAST>("", std::vector<std::string>());
    return std::make_unique<FunctionAST>(std::move(Proto), std::move(E));
  }
  return nullptr;
}
```

- 지금까지의 코드들을 실행할 수 있는 드라이버.

```
/// top ::= definition | external | expression | ';'
static void MainLoop() {
```

```
  while (true) {
    fprintf(stderr, "ready> ");
    switch (CurTok) {
    case tok_eof:
      return;
    case ';': // ignore top-level semicolons.
      getNextToken();
      break;
    case tok_def:
      HandleDefinition();
      break;
    case tok_extern:
      HandleExtern();
      break;
    default:
      HandleTopLevelExpression();
      break;
    }
  }
}
```

```
$ ./tut2          실행 예제
ready> def sum(a b) a+b;
[...]
Read function definition:define double @sum(double %a, double %b) {
entry:
  %addtmp = fadd double %a, %b
  ret double %addtmp
}

ready> extern sin(a);
[...]
Read extern: declare double @sin(double)

ready> 1+1;
[...]
Read top-level expression:define double @__anon_expr() {
entry:
  ret double 2.000000e+00
}
```

- 지금까지 렉서, 파서, AST 빌더를 포함하여 최소한의 언어를 정의하였음. 실행 파일은 Kal 코드를 검증하고 문법적으로 잘못될 경우 알려준다.

```
$ llvm-config --cxxflags --ldflags --system-libs --libs core
-I/usr/lib/llvm-18/include -std=c++17   -fno-exceptions -funwind-tables -D_GNU_SOURCE -D__STDC_
CONSTANT_MACROS -D__STDC_FORMAT_MACROS -D__STDC_LIMIT_MACROS
-L/usr/lib/llvm-18/lib
-lLLVM-18
$ clang++ -g -O3 tut2.cpp `llvm-config --cxxflags --ldflags --system-libs --libs core` -o tut2
$ ./tut2
ready> 1+2*3;     연산자 우선순위가 처리됨 ==> 결과는 7
[...]
Read top-level expression:define double @__anon_expr() {
entry:
```

```
  ret double 7.000000e+00
}
ready> (1+2)*3;      괄호에 의한 우선순위가 처리됨 ==> 결과는 9
[tut2]CurTok : 40
[...]
Read top-level expression:define double @__anon_expr() {
entry:
  ret double 9.000000e+00
}
ready> def sum(a b( a+b;     syntax 오류
[...]
Error: Expected ')' in prototype
[...]
ready> extern cos;       extern 함수는 '(' 가 있어야 함
[...]
Error: Expected '(' in prototype
```

3.3. IR 변환

- 생성된 AST를 LLVM IR코드로 변환하기 위해 각 AST 클래스에 codegen 가상 메서드 함수를 추가함. codegen() 메서드는 해당 AST 노드에 대한 IR을 종속된 모든 것과 함께 내보내라고 하며, 모두 LLVM Value 객체를 반환함. "Value"[26]는 LLVM에서 "정적 단일 할당(SSA) 레지스터" 또는 "SSA 값"[27]을 나타내는 데 사용되는 클래스

```
class ExprAST {
public:
  virtual ~ExprAST() = default;
  virtual Value *codegen() = 0;    // codegen 가상 함수 (파생클래스에서 재정의 됨)
};
class NumberExprAST : public ExprAST {
  double Val;
public:
  NumberExprAST(double Val) : Val(Val) {}
  Value *codegen() override;      // codegen 가상 함수를 재정의 함.
};
[...]
```

- LogErrorV 메서드는 codegen 과정에서 발생되는 에러를 보고하는 용도. TheContext는 타입 테이블, 상수값 테이블 등 LLVM에서 내부적으로 사용하는 데이터 구조를 소유하고 있는 객체임. Builder 는 LLVM 명령어를 쉽게 생성할 수 있는 도우미 객체임. IRBuilder 클래스[28]를 참조. TheModule [29]는 함수와 전역 변수를 포함하는 LLVM의 최상위 구조로 생성하는 모든 IR에 대한 메모리를 소유하게 되므로 codegen() 메서드는 unique_ptr<Value> 가 아닌 원시 Value*를 반환함. NamedValues 맵은 현재 범위에서 정의된 LLVM 표현(즉, 심볼 테이블)을 추적하는데 사용.

```
  codegen 과정에서 사용되는 정적 변수들
static std::unique_ptr<LLVMContext> TheContext;
static std::unique_ptr<IRBuilder<>> Builder(TheContext);
```

26) https://llvm.org/docs/doxygen/classllvm_1_1Value.html
27) https://en.wikipedia.org/wiki/Static_single-assignment_form
28) https://github.com/llvm/llvm-project/blob/main/llvm/include/llvm/IR/IRBuilder.h
29) https://github.com/llvm/llvm-project/blob/main/llvm/include/llvm/IR/Module.h

```
static std::unique_ptr<Module> TheModule;
static std::map<std::string, Value *> NamedValues;
Value *LogErrorV(const char *Str) {    // Value 타입을 리턴
  LogError(Str);
  return nullptr;
}
```

- 숫자표현식 노드에서 LLVM 코드를 생성. 숫자 상수는 ConstantFP 클래스[30]로 표현. 내부적으로 APFloat (arbtrary precision) 타입[31]의 값을 저장.

```
Value *NumberExprAST::codegen() {
  return ConstantFP::get(*TheContext, APFloat(Val));
}
```

- 변수에 대한 참조. 지정된 이름이 맵(NamedValues)에 있는지 확인하고 해당 값을 반환.

```
Value *VariableExprAST::codegen() {
  // Look this variable up in the function.
  Value *V = NamedValues[Name];
  if (!V)  LogErrorV("Unknown variable name");
  return V;
}
```

- 이진 연산자에 대한 IR생성. 표현식의 왼쪽에 대한 코드를 재귀적으로 내보내고, 오른쪽에 대한 코드를 내보내고, 이진 표현식의 결과를 계산. Builder 클래스는 값을 표시하기 시작함. IRBuilder에게 제공되는 이름 예를 들어 "addtmp"라는 이름으로 여러 코드에서 내보내는 경우 LLVM은 각각에 증가하는 고유한 숫자 접미사를 자동으로 제공함.

```
Value *BinaryExprAST::codegen() {
  Value *L = LHS->codegen();   // LHS, RHS는 클래스 내부 변수임
  Value *R = RHS->codegen();
  if (!L || !R)  return nullptr;

  switch (Op) {
  case '+':
    return Builder->CreateFAdd(L, R, "addtmp");
  case '-':
    return Builder->CreateFSub(L, R, "subtmp");
  case '*':
    return Builder->CreateFMul(L, R, "multmp");
  case '<':
    L = Builder->CreateFCmpULT(L, R, "cmptmp");
    // Convert bool 0/1 to double 0.0 or 1.0
    return Builder->CreateUIToFP(L, Type::getDoubleTy(TheContext),"booltmp");
  default:
    return LogErrorV("invalid binary operator");
  }
}
```

- 함수 호출 IR 생성

```
Value *CallExprAST::codegen() {
```

30) https://github.com/llvm/llvm-project/blob/main/llvm/lib/IR/Constants.cpp#L963
31) https://github.com/llvm/llvm-project/blob/main/llvm/include/llvm/ADT/APFloat.h

```
  // 모듈 객체의 테이블을 이용하여 함수 이름을 참조. Function[32] 객체를 리턴
  Function *CalleeF = TheModule->getFunction(Callee);
  if (!CalleeF)
    return LogErrorV("Unknown function referenced");

  // If argument mismatch error.
  if (CalleeF->arg_size() != Args.size())
    return LogErrorV("Incorrect # arguments passed");

  std::vector<Value *> ArgsV;
  for (unsigned i = 0, e = Args.size(); i != e; ++i) {
    ArgsV.push_back(Args[i]->codegen());
    if (!ArgsV.back())
      return nullptr;
  }

  return Builder->CreateCall(CalleeF, ArgsV, "calltmp");
}
```

- 함수 프로토타입 IR 생성.

```
  프로토타입은 Value* 대신 Function을 반환
Function *PrototypeAST::codegen() {
  // Make the function type:  double(double,double) etc.
  // Kal 언어에서는 double 타입만 사용하기 때문에 함수의 매개변수 개수만큼의 double 타입 벡터를
만든다.
  std::vector<Type *> Doubles(Args.size(), Type::getDoubleTy(*TheContext));
   사용될 FunctionType [33]을 반환. 위의 double 타입 벡터를 Parms 로 설정. isVarArg=false임
  FunctionType *FT = FunctionType::get(Type::getDoubleTy(*TheContext), Doubles, false);

  Function *F = Function::Create(FT, Function::ExternalLinkage, Name, TheModule.get());

  // 함수의 모든 매개 변수의 이름을 설정함
  unsigned Idx = 0;
  for (auto &Arg : F->args())
    Arg.setName(Args[Idx++]);

  return F;
}
```

- 함수 IR 생성. 함수 프로토타입은 함수의 본문(body)가 없는 반면, 함수는 본문이 있음.
  클래스 상속 관계 (Function <- GlobalObject <- GlobalValue <- Constant <- User <- Value)

```
Function *FunctionAST::codegen() {
  // 기존의 extern 선언으로부터 함수가 존재하는지 체크
  Function *TheFunction = TheModule->getFunction(Proto->getName());
  if (!TheFunction)  TheFunction = Proto->codegen();
  if (!TheFunction) return nullptr;

  // BasicBlock[34][35]을 생성하여 추가해 나감
  BasicBlock *BB = BasicBlock::Create(*TheContext, "entry", TheFunction);
  Builder->SetInsertPoint(BB);  // builder가 basic block의 insert() 메서드를 호출할 예정
```

32) https://github.com/llvm/llvm-project/blob/main/llvm/include/llvm/IR/Function.h
33) https://github.com/llvm/llvm-project/blob/main/llvm/include/llvm/IR/DerivedTypes.h#L104

```
  // 함수의 매개변수들을 NamedValues 맵에 추가
  NamedValues.clear();
  for (auto &Arg : TheFunction->args())
    NamedValues[std::string(Arg.getName())] = &Arg;

  if (Value *RetVal = Body->codegen()) {  // 함수 본문 Body는 클래스 내부 변수
    // Finish off the function.
    Builder->CreateRet(RetVal); // return 명령을 insert 함. RetVal은 body의 value가 된다.

    // Validate the generated code, checking for consistency.
    verifyFunction(*TheFunction);  // Verifier.cpp[36]에 정의됨
    return TheFunction;
  }
  // Error reading body, remove function.
  TheFunction->eraseFromParent();
  return nullptr;
}
```

- IR 생성 테스트. 몇 가지 입력에 대해서 IR 코드가 생성되는 것을 확인.

```
$ ./tut2
 최상위 표현식 처리 과정 : HandleTopLevelExpression -> ParseTopLevelExpr -> ParseExpression ->
ParsePrimary -> ParseNumberExpr -> ParseBinOpRHS
ready> 4+5;
Read top-level expression:
define double @0() {
entry:
  ret double 9.000000e+00
}

  함수 정의 HandleDefinition -> ParseDefinition -> ParsePrototype -> ParseExpression -> ...
ready> def foo(a b) a*a + 2*a*b + b*b;
Read function definition:
define double @foo(double %a, double %b) {
entry:
  %multmp = fmul double %a, %a               a*a
  %multmp1 = fmul double 2.000000e+00, %a    2*a
  %multmp2 = fmul double %multmp1, %b        2*a*b
  %addtmp = fadd double %multmp, %multmp2    a*a + 2*a*b
  %multmp3 = fmul double %b, %b              b*b
  %addtmp4 = fadd double %addtmp, %multmp3   a*a + 2*a*b + b*b
  ret double %addtmp4
}

  함수 선언 HandleExtern -> ParseExtern -> ParsePrototype
ready> extern cos(x);
ready> Read extern: declare double @cos(double)

  함수 호출 HandleTopLevelExpression -> ParseTopLevelExpr -> ParseExpression -> ParsePrimary ->
ParseParenExpr -> ...
```

34) https://github.com/llvm/llvm-project/blob/main/llvm/include/llvm/IR/BasicBlock.h
35) https://en.wikipedia.org/wiki/Basic_block
36) https://github.com/llvm/llvm-project/blob/main/llvm/lib/IR/Verifier.cpp#L7131

```
ready> cos(3.14);
ready> Read top-level expression:define double @__anon_expr() {
entry:
  %calltmp = call double @cos(double 3.140000e+00)
  ret double %calltmp
}
ready> ^D
 ; ModuleID = 'my cool jit'
source_filename = "my cool jit"
```

3.4. JIT 및 최적화 프로그램 지원추가

- 사소한 상수 접기(trivial constant folding)는 IRBuilder에서 간단한 코드를 컴파일할 때 명백한 최적화를 제공한다. LLVM은 공통 하위 표현식 제거(common sub-expression; CSE)와 같은 패스(pass) 형태로 사용할 수 있는 광범위한 최적화를 제공함.

```
$ ./tut2
ready> def test(x) 1+2+x;
[tut2]create new BasicBlock 'entry' 59
Read function definition:define double @test(double %x) {
entry:
  %addtmp = fadd double 3.000000e+00, %x    IRBuilder에 의해 상수 폴딩이 처리됨
  ret double %addtmp
}
ready> def test(x) (1+2+x)*(x+(1+2));    조금더 복잡한 예제. LHS와 RHS는 같음에 주의
[tut2]create new BasicBlock 'entry' 59
Read function definition:define double @test(double %x) {
entry:
  %addtmp = fadd double 3.000000e+00, %x     더하기를 2번 연산함. 최적화가 조금더 필요한 상황
  %addtmp1 = fadd double %x, 3.000000e+00      ==> " tmp = x+3; result = tmp*tmp "
  %multmp = fmul double %addtmp, %addtmp1
  ret double %multmp
}
```

- LLVM은 가능한 한 큰 코드 본문을 대상으로 하는 전체 모듈 패스와, 다른 함수를 살펴보지 않고 한번에 하나의 함수에서만 작동하는 함수별(per-function) 패스를 지원함. LLVM의 패스 작성 방법[37]과 패스 목록[38]을 참고한다.
- 분석 패스(analysis pass)는 다른 패스에서 사용할 수 있는 정보를 계산하며, 변환 패스(transform pass)는 IR을 변형하는 기능을 갖는다. 변환 패스를 추가하려면 해당 패스가 의존하는 모든 분석 패스를 미리 등록해야 함.
- 함수별 최적화를 진행하려면 실행하려는 LLVM 최적화를 보관하고 정리할 `FunctionPassManager`를 설정해야 함.

```
static void InitializeModuleAndManagers() {
  // Open a new context and module.
  TheContext = std::make_unique<LLVMContext>();
  TheModule = std::make_unique<Module>("KaleidoscopeJIT", *TheContext);
  TheModule->setDataLayout(TheJIT->getDataLayout());   // JIT 데이터 레이아웃을 설정
```

37) https://llvm.org/docs/WritingAnLLVMPass.html
38) https://llvm.org/docs/Passes.html

```
  // Create a new builder for the module.
  Builder = std::make_unique<IRBuilder<>>(*TheContext);

  // 패스를 생성함. 4개의 분석 관리자를 생성함
  TheFPM = std::make_unique<FunctionPassManager>();
  TheLAM = std::make_unique<LoopAnalysisManager>();  // 루프
  TheFAM = std::make_unique<FunctionAnalysisManager>(); // 함수
  TheCGAM = std::make_unique<CGSCCAnalysisManager>();  // call graph strongly connected component
  TheMAM = std::make_unique<ModuleAnalysisManager>();  // 모듈
  ThePIC = std::make_unique<PassInstrumentationCallbacks>();
  TheSI = std::make_unique<StandardInstrumentations>(*TheContext, /*DebugLogging*/ true);
  TheSI->registerCallbacks(*ThePIC, TheMAM.get());

  // Add transform passes.
    CombinePass: multiple instruction을 단순화 한다. CFG를 변경하지는 않음.
%Y = add i32 %X, 1    (y=x+1)
%Z = add i32 %Y, 1    (z=y+1)
==>
%Z = add i32 %X, 2    (z=x+2)
  TheFPM->addPass(InstCombinePass());

  // Reassociate expressions.
 예를 들어 "(A + B) + C" 를 "(A + C) + B" 로 재배치하여 constant folding을 가능하게 하거나 단
순화가 가능하게 재배치함
  TheFPM->addPass(ReassociatePass());

  // Eliminate Common SubExpressions.
 GVN(global value numbering)은 동일한 표현식을 확인하여 중복적인 연산이 없도록 한다.
 동일한 값을 계산하는 명령들을 추적한다.
  TheFPM->addPass(GVNPass());

  // Simplify the control flow graph (deleting unreachable blocks, etc).
 basic block을 merge하더나, branch 단순화, switch 문을 단순화, unreachable code 제거,
  phi 노드의 제거 등을 수행하여 CGF(제어 흐름 그래프)를 단순화함.
  TheFPM->addPass(SimplifyCFGPass());

  // 변환 패스에서 사용된 분석 패스들을 등록함
  PassBuilder PB;
  PB.registerModuleAnalyses(*TheMAM);
  PB.registerFunctionAnalyses(*TheFAM);
  PB.crossRegisterProxies(*TheLAM, *TheFAM, *TheCGAM, *TheMAM);
}
```

- PassManager는 FunctionAST::codegen()에서 다음과 같이 활용된다.

```
Function *FunctionAST::codegen() {
 [...]
  if (Value *RetVal = Body->codegen()) {
    // Finish off the function.
    Builder->CreateRet(RetVal);
    // Validate the generated code, checking for consistency.
    verifyFunction(*TheFunction);

    // 함수에 대한 최적화를 수행
    TheFPM->run(*TheFunction, *TheFAM);
```

```
    return TheFunction;
  }

  // Error reading body, remove function.
  TheFunction->eraseFromParent();
  return nullptr;
}
```

```
 함수 최적화가 작동하는지 테스트
$ clang++ -g -O3 `llvm-config --cxxflags --ldflags --system-libs --libs core orcjit native` \
  tut3.cpp  -o tut3
$ ./tut3
ready> def test(x) (1+2+x)*(x+(1+2));
ready> Read function definition:define double @test(double %x) {
entry:
  %addtmp = fadd double %x, 3.000000e+00    덧셈이 2번이 아니라 1번만 수행됨
  %multmp = fmul double %addtmp, %addtmp
  ret double %multmp
}
```

- JIT(just-in-time) 컴파일러 추가. Kal 언어에서는 입력한 최상위 표현식을 즉시 평가함. 예를 들어 "1+2;"를 입력하면 3을 평가하고 출력하거나, 함수를 정의하고 명령줄에서 호출할 수 있어야 함. 네이티브 타겟에 대한 초기화가 필요함

```
static std::unique_ptr<KaleidoscopeJIT> TheJIT;  // simple JIT for Kal [39]
[...]
int main() {
  InitializeNativeTarget();   // TargetSelect.h 참고 [40]
  InitializeNativeTargetAsmPrinter();
  InitializeNativeTargetAsmParser();
[...]
  TheJIT = ExitOnErr(KaleidoscopeJIT::Create());
  InitializeModuleAndManagers();

  // Run the main "interpreter loop" now.
  MainLoop();
  return 0;
}
```

- KaleidoscopeJIT 클래스는 이 튜토리얼을 위해 특별히 빌드된 간단한 JIT 클래스임. addModule 메서드는 JIT에 LLVM IR 모듈을 추가하여 함수를 실행할 수 있게 함. (실행시 메모리는 ResourceTracker로 관리됨) lookup 메서드는 컴파일된 코드에 대한 포인터를 찾을 수 있게 한다.

```
// KaleidoscopeJIT.h
class KaleidoscopeJIT {
private:
  std::unique_ptr<ExecutionSession> ES;
[...]
  Error addModule(ThreadSafeModule TSM, ResourceTrackerSP RT = nullptr) {
    if (!RT)      RT = MainJD.getDefaultResourceTracker();
    return CompileLayer.add(RT, std::move(TSM));
  }
```

39) https://github.com/llvm/llvm-project/blob/main/llvm/examples/Kaleidoscope/include/KaleidoscopeJIT.h
40) https://github.com/llvm/llvm-project/blob/main/llvm/include/llvm/Support/TargetSelect.h

```
  Expected<ExecutorSymbolDef> lookup(StringRef Name) {
    return ES->lookup({&MainJD}, Mangle(Name.str()));
  }
```

- 이 간단한 API를 사용하여 최상위 표현식을 구문 분석하는 코드를 다음과 같이 변경함 :

```
static void HandleTopLevelExpression() {
  // Evaluate a top-level expression into an anonymous function.
  if (auto FnAST = ParseTopLevelExpr()) {
    if (FnAST->codegen()) {
      // Create a ResourceTracker to track JIT'd memory allocated to our
      // anonymous expression -- that way we can free it after executing.
      auto RT = TheJIT->getMainJITDylib().createResourceTracker();

      auto TSM = ThreadSafeModule(std::move(TheModule), std::move(TheContext));
      ExitOnErr(TheJIT->addModule(std::move(TSM), RT));
      InitializeModuleAndManagers();

      // __anon_expr 심볼을 찾음. ParseTopLevelExpr()에서 anonymous proto 로 만들어짐
      auto ExprSymbol = ExitOnErr(TheJIT->lookup("__anon_expr"));

      // Get the symbol's address and cast it to the right type (takes no
      // arguments, returns a double) so we can call it as a native function.
      double (*FP)() = ExprSymbol.getAddress().toPtr<double (*)()>();
      fprintf(stderr, "Evaluated to %f\n", FP());

      // Delete the anonymous expression module from the JIT.
      ExitOnErr(RT->remove());
    }
  } else {
    getNextToken();
  }
}
```

- HandleDefinition 과 HandleExtern 함수도 다음과 같이 수정. FunctionProtos는 각 함수의 최신 프로토타입을 보관하는 글로벌 맵이다. codegen()에서 TheModule->getFunction() 호출을 대체하기 위한 getFunction() 편의 함수를 정의.

```
static void HandleDefinition() {
  if (auto FnAST = ParseDefinition()) {
    if (auto *FnIR = FnAST->codegen()) {
      fprintf(stderr, "Read function definition:");
      FnIR->print(errs());
      fprintf(stderr, "\n");
      ExitOnErr(TheJIT->addModule(
          ThreadSafeModule(std::move(TheModule), std::move(TheContext))));
      InitializeModuleAndManagers();
[...]
static void HandleExtern() {
  if (auto ProtoAST = ParseExtern()) {
    if (auto *FnIR = ProtoAST->codegen()) {
      fprintf(stderr, "Read extern: ");
      FnIR->print(errs());
```

```
      fprintf(stderr, "\n");
      FunctionProtos[ProtoAST->getName()] = std::move(ProtoAST);
    }
[...]
Function *getFunction(std::string Name) {
  // 먼저 현재 모듈에서 기존 함수의 선언을 검색
  if (auto *F = TheModule->getFunction(Name))
    return F;

  // 만약 찾지 못했으면, FunctionProtos 에서 새 선언을 생성함
  auto FI = FunctionProtos.find(Name);
  if (FI != FunctionProtos.end())
    return FI->second->codegen();

  // If no existing prototype exists, return null.
  return nullptr;
}
```

- 어떻게 작동하는지 확인하기

```
$ clang++ -g `llvm-config --cxxflags --ldflags --system-libs --libs core orcjit native` -O3 \
  tut3.cpp  -o tut3
$ ./tut3
ready> 10-12;
ready> Read top-level expression:define double @__anon_expr() {
entry:
  ret double -2.000000e+00
}
Evaluated to -2.000000
ready> def testfunc(x y) x + y*2;
ready> Read function definition:define double @testfunc(double %x, double %y) {
entry:
  %multmp = fmul double %y, 2.000000e+00    함수 정의 testfunc(x y) => y*2 + x
  %addtmp = fadd double %x, %multmp
  ret double %addtmp
}

ready> testfunc(1,2);
ready> Read top-level expression:define double @__anon_expr() {
entry:                                      함수 호출 testfunc(1, 2)
  %calltmp = call double @testfunc(double 1.000000e+00, double 2.000000e+00)
  ret double %calltmp
}
Evaluated to 5.000000          x=1, y=2 ==> y*2 + x ==> 2*2+1 ==> 5
ready> extern sin(x);
ready> Read extern: declare double @sin(double)
ready> extern cos(x);
ready> Read extern: declare double @cos(double)
ready> def foo(x) sin(x)*sin(x) + cos(x)*cos(x);
ready> Read function definition:define double @foo(double %x) {
entry:
  %calltmp = call double @sin(double %x)
  %calltmp1 = call double @sin(double %x)     sin,cos 함수를 2번 호출함. 최적화는 되지 않음
  %multmp = fmul double %calltmp, %calltmp1
  %calltmp2 = call double @cos(double %x)
```

```
  %calltmp3 = call double @cos(double %x)
  %multmp4 = fmul double %calltmp2, %calltmp3
  %addtmp = fadd double %multmp, %multmp4
  ret double %addtmp
}
ready> foo(4.0);    (※ 수학적으로 항상 sin^2 + cos^2 = 1 임)
Read top-level expression:define double @__anon_expr() {
entry:
  %calltmp = call double @foo(double 4.000000e+00)
  ret double %calltmp
}
Evaluated to 1.000000        계산된 결과
```

3.5. 확장하기: 제어 흐름

- 본 절에서는 if/then/else 문과, 간단한 for 루프문을 지원하도록 Kal언어를 확장한다.
- 먼저 if/then/else를 적용한 예제와 이것을 컴파일한 결과를 먼저 살펴본다.

```
def fib(x)          피보나치 수열  1,1,2,3,5,8,13,...
  if x < 3 then    1
  else    fib(x-1)+fib(x-2);
ready> def fib(x)
  if x < 3 then    1
  else    fib(x-1)+fib(x-2);ready>
Read function definition:define double @fib(double %x) {
entry:
  %cmptmp = fcmp ult double %x, 3.000000e+00       x 와 3.0 을 비교. ult : unordered less than
  br i1 %cmptmp, label %ifcont, label %else

else:                                             ; preds = %entry
  %subtmp = fadd double %x, -1.000000e+00
  %calltmp = call double @fib(double %subtmp)
  %subtmp1 = fadd double %x, -2.000000e+00
  %calltmp2 = call double @fib(double %subtmp1)
  %addtmp = fadd double %calltmp, %calltmp2
  br label %ifcont

ifcont:                                           ; preds = %entry, %else
     phi[41] 명령은 여러 기본 블럭에서 오는 값들을 병합한다.
  %iftmp = phi double [ %addtmp, %else ], [ 1.000000e+00, %entry ]
  ret double %iftmp
}
```

- 렉서에 토큰을 추가하고, gettok() 토크나이져 함수에 추가함.

```
// control
tok_if = -6,
tok_then = -7,
tok_else = -8,
```

```
    if (IdentifierStr == "if")
      return tok_if;
    if (IdentifierStr == "then")
```

41) https://llvm.org/docs/LangRef.html#phi-instruction

```
      return tok_then;
    if (IdentifierStr == "else")
      return tok_else;
```

- AST를 확장하기 위해 IfExprAST 클래스를 추가한다.

```
/// IfExprAST - Expression class for if/then/else.
class IfExprAST : public ExprAST {
  std::unique_ptr<ExprAST> Cond, Then, Else;

public:
  IfExprAST(std::unique_ptr<ExprAST> Cond, std::unique_ptr<ExprAST> Then,
            std::unique_ptr<ExprAST> Else)
      : Cond(std::move(Cond)), Then(std::move(Then)), Else(std::move(Else)) {}

  Value *codegen() override;
};
```

- 파서 함수를 작성한다.

```
/// ifexpr ::= 'if' expression 'then' expression 'else' expression
static std::unique_ptr<ExprAST> ParseIfExpr() {
  getNextToken(); // eat the if.

  // condition.
  auto Cond = ParseExpression();
  if (!Cond)    return nullptr;

  if (CurTok != tok_then)
    return LogError("expected then");
  getNextToken(); // eat the then

  auto Then = ParseExpression();
  if (!Then)    return nullptr;

  if (CurTok != tok_else)    return LogError("expected else");

  getNextToken();
  auto Else = ParseExpression();
  if (!Else)    return nullptr;
  return std::make_unique<IfExprAST>(std::move(Cond), std::move(Then), std::move(Else));
}
```

- 주표현식에 포함시켜준다.

```
static std::unique_ptr<ExprAST> ParsePrimary() {
  switch (CurTok) {
  default:
    return LogError("unknown token when expecting an expression");
  case tok_identifier:
    return ParseIdentifierExpr();
  case tok_number:
    return ParseNumberExpr();
  case '(':
    return ParseParenExpr();
```

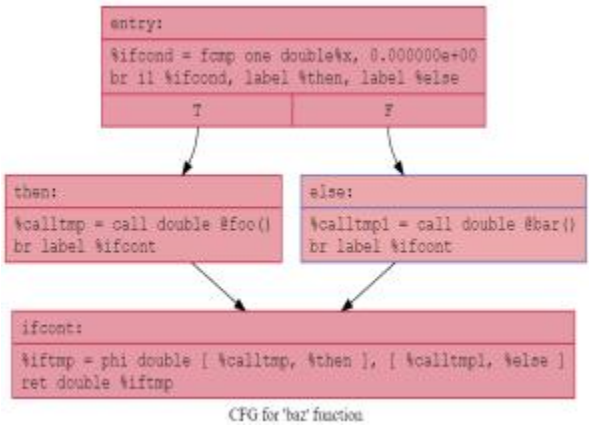
```
  case tok_if:
    return ParseIfExpr();
  }
}
```

- 다음 단계인 IR 코드를 생성하기 전에 LLVM의 **opt**[42] 툴을 이용하여 CFG(Context-Free Grammer)[43] 를 가시화하는 방법에 대해서 알아본다. Kal언어 **baz.kal**는 IR 코드 **baz.ll**로 변환된다. 이것을 **opt** 툴을 이용하여 graphviz(**.dot**) 파일로 변환할 수 있다.

```
$ more baz.kal
extern foo();
extern bar();
def baz(x) if x then foo() else bar();
```

```
$ more baz.ll
declare double @foo()
declare double @bar()
define double @baz(double %x) {
entry:
  %ifcond = fcmp one double %x, 0.000000e+00
  br i1 %ifcond, label %then, label %else

then:       ; preds = %entry
  %calltmp = call double @foo()
  br label %ifcont

else:       ; preds = %entry
  %calltmp1 = call double @bar()
  br label %ifcont

ifcont:     ; preds = %else, %then
 phi 명령은 SSA 형식을 구현하는 역할 44). phi 노드는 두 값을 병합하는 역할을 함
   제어 흐름이 then 에서 오는 경우 iftmp는 calltmp의 값을 가져옴.
   제어 흐름이 else 에서 오는 경우 iftmp는 calltmp1의 값을 가져옴
  %iftmp = phi double [ %calltmp, %then ], [ %calltmp1, %else ]
  ret double %iftmp
}
```

```
$ opt -print-passes   이용가능한 pass들을 출력
[...]
dot-cfg

$ llvm-as < fib.ll | opt -passes=dot-cfg
   dot-cfg 패스를 이용하여 .dot 파일을 생성
[...]
Writing '.baz.dot'...

$ more .baz.dot
    dot 파일을 GraphViz Online45)에 입력
digraph "CFG for 'baz' function" {
        label="CFG for 'baz' function";
        Node0x5dc1b237ae80 [shape=record,color="#b70
d28ff", style=filled, fillcolor="#b70d2870" fontname
="Courier",label="{entry:\l|  %ifcond = fcmp one dou
ble
%x, 0.000000e+00\l  br i1 %ifcond, label %then, labe
l %else\l|{<s0>T|<s1>F}}"];
```



CFG for 'baz' function

42) https://llvm.org/docs/CommandGuide/opt.html
43) https://en.wikipedia.org/wiki/Context-free_grammar

```
[...]
}
```

- if/then/else IR 코드 생성

```
Value *IfExprAST::codegen() {
  Value *CondV = Cond->codegen();
  if (!CondV)    return nullptr;

  // Convert condition to a bool by comparing non-equal to 0.0.
  CondV = Builder->CreateFCmpONE(    floating-point 비교, ONE:ordered and not equal
      CondV, ConstantFP::get(*TheContext, APFloat(0.0)), "ifcond");

  Function *TheFunction = Builder->GetInsertBlock()->getParent();

  // Create blocks for the then and else cases.  Insert the 'then' block at the
  // end of the function.
  BasicBlock *ThenBB = BasicBlock::Create(*TheContext, "then", TheFunction);
  BasicBlock *ElseBB = BasicBlock::Create(*TheContext, "else");
  BasicBlock *MergeBB = BasicBlock::Create(*TheContext, "ifcont");
  Builder->CreateCondBr(CondV, ThenBB, ElseBB);  // conditional branch 생성

  // Emit then value.
  Builder->SetInsertPoint(ThenBB);
  Value *ThenV = Then->codegen();
  if (!ThenV)    return nullptr;

  Builder->CreateBr(MergeBB);
  // Codegen of 'Then' can change the current block, update ThenBB for the PHI.
  ThenBB = Builder->GetInsertBlock();

  // Emit else block.
  TheFunction->insert(TheFunction->end(), ElseBB);
  Builder->SetInsertPoint(ElseBB);
  Value *ElseV = Else->codegen();
  if (!ElseV)    return nullptr;

  Builder->CreateBr(MergeBB);
  // Codegen of 'Else' can change the current block, update ElseBB for the PHI.
  ElseBB = Builder->GetInsertBlock();

  // Emit merge block.
  TheFunction->insert(TheFunction->end(), MergeBB);
  Builder->SetInsertPoint(MergeBB);
  PHINode *PN = Builder->CreatePHI(Type::getDoubleTy(*TheContext), 2, "iftmp");
  PN->addIncoming(ThenV, ThenBB);
  PN->addIncoming(ElseV, ElseBB);
  return PN;
}
```

- 조건문 테스트

44) https://llvm.org/docs/LangRef.html#phi-instruction
45) https://dreampuf.github.io/GraphvizOnline/

```
# fibinacci number
def fib(x) if x < 3 then 1   #comment
  else fib(x-1)+fib(x-2);
```

```
$ ./tut5
ready> def fib(x) if x < 3 then 1
  else fib(x-1)+fib(x-2);ready>
Read function definition:define double @fib(double %x) {
[...]
ready> fib(10);
ready> Evaluated to 55.000000
ready> fib(20);
ready> Evaluated to 6765.000000
ready> fib(30);
ready> Evaluated to 832040.000000
```

- if/then/else 문에 이어서 for 루프 표현식에 대해서 다루기 전에 예제를 먼저 살펴본다.

```
$ more star.h
extern void star(void);
$ more star.c
#include <stdio.h>
void star(void) {
  puts("*");     // '*' 문자를 출력하는 외부 함수 star()
}
    공유 라이브러리로 컴파일 하기
gcc -c -fpic star.c
gcc -shared -o libstar.so star.o
   libstar를 포함하여 컴파일 하기 (-lstar) (tut5.c)
$ clang++ -g -O3 `llvm-config --cxxflags --ldflags --system-libs --libs core orcjit native` \
    tut5.cpp  -o tut5  -lstar -L.
$ LD_LIBRARY_PATH=.   ./tut5
```

```
extern star();
def printstar(n)           for 루프를 이용하여 n개의 문자를 출력
  for i = 1, i < n, 1.0 in
    star();
printstar(10);
```

```
ready> extern star();
ready> Read extern: declare double @star()

ready> def printstar(n)
  for i = 1, i < n, 1.0 in
    star();ready>
Read function definition:define double @printstar(double %n) {
entry:
  br label %loop

loop:                                             ; preds = %loop, %entry
  %i = phi double [ 1.000000e+00, %entry ], [ %nextvar, %loop ]
  %calltmp = call double @star()                    함수를 호출
  %nextvar = fadd double %i, 1.000000e+00           1을 증가
  %cmptmp = fcmp ult double %i, %n                  비교 %i < %n
  br i1 %cmptmp, label %loop, label %afterloop      루프를 반복

afterloop:                                        ; preds = %loop
  ret double 0.000000e+00
```

```
}
ready> printstar(10);
ready> Evaluated to 0.000000
ready> **********  <-------- 문자가 10개 출력됨
```

- 렉서와 토크나이저에 추가하기

```
   tok_for = -9, tok_in = -10
    if (IdentifierStr == "for")
      return tok_for;
    if (IdentifierStr == "in")
      return tok_in;
```

- for 루프 AST 확장

```
/// ForExprAST - Expression class for for/in.
class ForExprAST : public ExprAST {
  std::string VarName;   // 변수명
  std::unique_ptr<ExprAST> Start, End, Step, Body;  // 표현식 조각들

public:
  ForExprAST(const std::string &VarName, std::unique_ptr<ExprAST> Start,
             std::unique_ptr<ExprAST> End, std::unique_ptr<ExprAST> Step,
             std::unique_ptr<ExprAST> Body)
      : VarName(VarName), Start(std::move(Start)), End(std::move(End)),
        Step(std::move(Step)), Body(std::move(Body)) {}

  Value *codegen() override;
};
```

- 파서 확장

```
/// forexpr ::= 'for' identifier '=' expr ',' expr (',' expr)? 'in' expression
static std::unique_ptr<ExprAST> ParseForExpr() {
  getNextToken();  // eat the for.

  if (CurTok != tok_identifier)
    return LogError("expected identifier after for");

  std::string IdName = IdentifierStr;
  getNextToken();  // eat identifier.

  if (CurTok != '=')    return LogError("expected '=' after for");
  getNextToken();  // eat '='.

  auto Start = ParseExpression();
  if (!Start)    return nullptr;
  if (CurTok != ',')    return LogError("expected ',' after for start value");
  getNextToken();

  auto End = ParseExpression();
  if (!End)    return nullptr;

  // The step value is optional.
  std::unique_ptr<ExprAST> Step;
```

```
  if (CurTok == ',') {
    getNextToken();
    Step = ParseExpression();
    if (!Step)      return nullptr;
  }

  if (CurTok != tok_in)    return LogError("expected 'in' after for");
  getNextToken();  // eat 'in'.

  auto Body = ParseExpression();
  if (!Body)    return nullptr;

  return std::make_unique<ForExprAST>(IdName, std::move(Start),
                 std::move(End), std::move(Step), std::move(Body));
}
static std::unique_ptr<ExprAST> ParsePrimary() {
[...]
  case tok_for:
    return ParseForExpr();
```

- for문을 위한 IR 생성

```
Value *ForExprAST::codegen() {
  // Emit the start code first, without 'variable' in scope.
  Value *StartVal = Start->codegen();
  if (!StartVal)    return nullptr;

  // loop header 를 위한 새로운 basic block을 만들기. 현재 블럭 뒤에 삽입함
  Function *TheFunction = Builder->GetInsertBlock()->getParent();
  BasicBlock *PreheaderBB = Builder->GetInsertBlock();
  BasicBlock *LoopBB = BasicBlock::Create(*TheContext, "loop", TheFunction);

  // Insert an explicit fall through from the current block to the LoopBB.
  Builder->CreateBr(LoopBB);   무조건 분기 명령을 생성

  // Start insertion in LoopBB.
  Builder->SetInsertPoint(LoopBB);

  // PHI 노드를 생성. 입력값의 최대 개수는 2개. VarName:생성된 phi 노드의 이름
  PHINode *Variable = Builder->CreatePHI(Type::getDoubleTy(*TheContext), 2, VarName);
  // phi 노드에 입력값을 추가. 초기값:StartVal
  Variable->addIncoming(StartVal, PreheaderBB);

  // for 루프는 심볼 테이블에 새 변수를 추가함. 루프 본문 코드를 생성하기 전에 루프 변수를 추가
     함. 외부 범위에 같은 이름의 변수가 있을 수 있음에 유의. 변수 섀도잉[46]을 허용함. 섀도잉할
     변수를 OldVal에 저장. 섀도잉할 변수가 없는 경우 OldVal은 null이 됨.
  Value *OldVal = NamedValues[VarName];
  NamedValues[VarName] = Variable;

  // 루프 본문에 대한 코드를 재귀적으로 생성함
  if (!Body->codegen())   return nullptr;

  // Emit the step value.
  Value *StepVal = nullptr;
  if (Step) {
```

```
    StepVal = Step->codegen();
    if (!StepVal)      return nullptr;
  } else {
    // 반복 변수의 다음 값을 계산, 값이 없으면 1.0을 계산함
    StepVal = ConstantFP::get(*TheContext, APFloat(1.0));
  }
  Value *NextVar = Builder->CreateFAdd(Variable, StepVal, "nextvar");

  // 루프의 종료값을 평가하여 루프를 종료해야 하는지 여부를 결정
  Value *EndCond = End->codegen();
  if (!EndCond)    return nullptr;

  // Convert condition to a bool by comparing non-equal to 0.0.
  EndCond = Builder->CreateFCmpONE(
      EndCond, ConstantFP::get(*TheContext, APFloat(0.0)), "loopcond");

  // Create the "after loop" block and insert it.
  BasicBlock *LoopEndBB = Builder->GetInsertBlock();
  BasicBlock *AfterBB = BasicBlock::Create(*TheContext, "afterloop", TheFunction);

  // Insert the conditional branch into the end of LoopEndBB.
  Builder->CreateCondBr(EndCond, LoopBB, AfterBB);

  // Any new code will be inserted in AfterBB.
  Builder->SetInsertPoint(AfterBB);

  // Add a new entry to the PHI node for the backedge.
  Variable->addIncoming(NextVar, LoopEndBB);

  // Restore the unshadowed variable.
  if (OldVal)    NamedValues[VarName] = OldVal;
  else    NamedValues.erase(VarName);

  // for expr always returns 0.0.
  return Constant::getNullValue(Type::getDoubleTy(*TheContext));
}
```

```
$ more start.ll    다음과 같은 IR 코드가 생성된다.
declare double @star()

define double @printstar(double %n) {
entry:
  br label %loop

loop:                                             ; preds = %loop, %entry
  %i = phi double [ 1.000000e+00, %entry ], [ %nextvar, %loop ]
  %calltmp = call double @star()
  %nextvar = fadd double %i, 1.000000e+00
  %cmptmp = fcmp ult double %i, %n
  br i1 %cmptmp, label %loop, label %afterloop

afterloop:                                        ; preds = %loop
  ret double 0.000000e+00
}
```

- for 루프 테스트

```
#  문자를 반복해서 출력하기
extern star();
def printstar(n)
  for i = 1, i < n, 1.0 in  star();
printstar(10); printstar(20);
```

```
ready> extern star();
ready> Read extern: declare double @star()

ready> def printstar(n)
  for i = 1, i < n, 1.0 in  star();ready>
Read function definition:define double @printstar(double %n) {
[...]
ready> printstar(20);
ready> Evaluated to 0.000000
ready> ********************
```

3.6. 확장하기: 사용자 정의 연산자

- 본 절에서는 유용한 연산자(나누기, 논리 부정, 비교)에 대한 지원을 추가한다. 연산자 오버로딩(operator overloading)은 C++과 같은 언어에서 지원하지만, C++에서는 기존의 연산자만 재정의 할 수 있고, 새로운 연산자를 도입하거나 우선순위를 변경할 수 없음.
- 추가할 연산자는 논리부정(!) 단항 연상자와, 비교(>), 논리OR(|), 비교(=) 임.

```
# Logical unary not.
def unary!(v)
  if v then    0
  else    1;

# Define > with the same precedence as <. 우선순위는 10
def binary> 10 (LHS RHS)
  RHS < LHS;

# Binary "logical or", (note that it does not "short circuit") 우선순위는 5
def binary| 5 (LHS RHS)
  if LHS then    1
  else if RHS then    1
  else    0;

# Define = with slightly lower precedence than relationals. 우선순위는 9
def binary= 9 (LHS RHS)
  !(LHS < RHS | LHS > RHS);
```

- 단항/이진 키워드에 대한 지원을 추가

```
  // operators
  tok_binary = -11,
  tok_unary = -12
```

```
static int gettok() {
[...]
    if (IdentifierStr == "binary")
```

46) https://en.wikipedia.org/wiki/Variable_shadowing

```
      return tok_binary;
    if (IdentifierStr == "unary")
      return tok_unary;
```

- 예를 들어 "def binary¦ 5" 는 새로운 연산자의 정의를 표현한다. 이것은 프로토타입 생성으로 구문 분석되어 AST 노드로 들어간다.

```
class PrototypeAST {
  std::string Name;
  std::vector<std::string> Args;
  bool IsOperator;
  unsigned Precedence;  // 이진연산자의 경우 우선순위

public:
  PrototypeAST(const std::string &Name, std::vector<std::string> Args,
               bool IsOperator = false, unsigned Prec = 0)
  : Name(Name), Args(std::move(Args)), IsOperator(IsOperator), Precedence(Prec) {}

  Function *codegen();
  const std::string &getName() const { return Name; }

  bool isUnaryOp() const { return IsOperator && Args.size() == 1; }
  bool isBinaryOp() const { return IsOperator && Args.size() == 2; }
  char getOperatorName() const {
    assert(isUnaryOp() ¦¦ isBinaryOp());
    return Name[Name.size() - 1];
  }
  unsigned getBinaryPrecedence() const { return Precedence; }
};
```

- 구문 분석 하기

```
/// prototype
///   ::= id '(' id* ')'
///   ::= binary LETTER number? (id, id)
static std::unique_ptr<PrototypeAST> ParsePrototype() {
  std::string FnName;

  unsigned Kind = 0;  // 0 = identifier, 1 = unary, 2 = binary.
  unsigned BinaryPrecedence = 30;

  switch (CurTok) {
  default:
    return LogErrorP("Expected function name in prototype");
  case tok_identifier:
    FnName = IdentifierStr;
    Kind = 0;
    getNextToken();
    break;
  case tok_binary:
    getNextToken();
    if (!isascii(CurTok)) return LogErrorP("Expected binary operator");
    FnName = "binary";
    FnName += (char)CurTok;    이진 연산자에 대한 이름 “binary¦" 와 같은 이름
    Kind = 2;
```

```
    getNextToken();

    // Read the precedence if present.
    if (CurTok == tok_number) {
      if (NumVal < 1 || NumVal > 100)
        return LogErrorP("Invalid precedence: must be 1..100");
      BinaryPrecedence = (unsigned)NumVal;
      getNextToken();
    }
    break;
  }
[...]

  // Verify right number of names for operator.
  if (Kind && ArgNames.size() != Kind)
    return LogErrorP("Invalid number of operands for operator");
  return std::make_unique<PrototypeAST>(FnName, std::move(ArgNames), Kind != 0,
                                        BinaryPrecedence);
}
```

- IR 코드 생성

```
Value *BinaryExprAST::codegen() {
  Value *L = LHS->codegen();
  Value *R = RHS->codegen();
  if (!L || !R)    return nullptr;

  switch (Op) {
  case '+':
    return Builder->CreateFAdd(L, R, "addtmp");
  case '-':
    return Builder->CreateFSub(L, R, "subtmp");
  case '*':
    return Builder->CreateFMul(L, R, "multmp");
  case '<':
    L = Builder->CreateFCmpULT(L, R, "cmptmp");
    // Convert bool 0/1 to double 0.0 or 1.0
    return Builder->CreateUIToFP(L, Type::getDoubleTy(*TheContext), "booltmp");
  default:
    break;
  }

  // If it wasn't a builtin binary operator, it must be a user defined one. Emit
  // a call to it.
  Function *F = getFunction(std::string("binary") + Op);
  assert(F && "binary operator not found!");

  Value *Ops[2] = { L, R };
  return Builder->CreateCall(F, Ops, "binop");
}
```

- 사용자 정의 단항 연산자 AST

```
/// UnaryExprAST - Expression class for a unary operator.
class UnaryExprAST : public ExprAST {
```

```
  char Opcode;
  std::unique_ptr<ExprAST> Operand;

public:
  UnaryExprAST(char Opcode, std::unique_ptr<ExprAST> Operand)
    : Opcode(Opcode), Operand(std::move(Operand)) {}

  Value *codegen() override;
};
```

- 단항 연산자 정의 파서

```
/// unary
///   ::= primary
///   ::= '!' unary
static std::unique_ptr<ExprAST> ParseUnary() {
  // If the current token is not an operator, it must be a primary expr.
  if (!isascii(CurTok) || CurTok == '(' || CurTok == ',')
    return ParsePrimary();

  // If this is a unary operator, read it.
  int Opc = CurTok;
  getNextToken();
  if (auto Operand = ParseUnary())
    return std::make_unique<UnaryExprAST>(Opc, std::move(Operand));
  return nullptr;
}
```

- 주표현식에서 단항 연산자를 처리할 수 있도록 함.

```
/// binoprhs
///   ::= ('+' unary)*
static std::unique_ptr<ExprAST> ParseBinOpRHS (int ExprPrec,
                                               std::unique_ptr<ExprAST> LHS) {
[...]
    // Parse the unary expression after the binary operator.
    auto RHS = ParseUnary();   // ParsePrimary()에서  ParseUnary()로 변경
    if (!RHS)      return nullptr;
```

```
/// expression
///   ::= unary binoprhs
///
static std::unique_ptr<ExprAST> ParseExpression() {
  auto LHS = ParseUnary();  // ParsePrimary()에서  ParseUnary()로 변경
  if (!LHS)    return nullptr;

  return ParseBinOpRHS(0, std::move(LHS));
}
```

- 파서에서 단항 연산자 지원을 추가

```
/// prototype
///   ::= id '(' id* ')'
///   ::= binary LETTER number? (id, id)
///   ::= unary LETTER (id)
```

```
static std::unique_ptr<PrototypeAST> ParsePrototype() {
  std::string FnName;

  unsigned Kind = 0;  // 0 = identifier, 1 = unary, 2 = binary.
  unsigned BinaryPrecedence = 30;

  switch (CurTok) {
  default:
    return LogErrorP("Expected function name in prototype");
[...]
  case tok_unary:
    getNextToken();
    if (!isascii(CurTok))  return LogErrorP("Expected unary operator");
    FnName = "unary";
    FnName += (char)CurTok;  단항 연산자에 대한 이름 “unary!" 와 같은 이름
    Kind = 1;
    getNextToken();
    break;
  case tok_binary:
    ...
```

- 단항 연산자 IR 생성

```
Value *UnaryExprAST::codegen() {
  Value *OperandV = Operand->codegen();
  if (!OperandV)    return nullptr;

  Function *F = getFunction(std::string("unary") + Opcode);
  if (!F)    return LogErrorV("Unknown unary operator");

  return Builder->CreateCall(F, OperandV, "unop");
}
```

- 지금까지 확장한 Kal 언어를 이용하여 mandelbrot set [47] 프로그램 테스트가 가능하다

```
   비교 연산자가 작동하는지 테스트
def great(a b) a > b;
def less(a b) a < b;
ready> 1<2;
ready> Evaluated to 1.000000    참(ture)
ready> 1>2;
ready> Evaluated to 0.000000    거짓(false)
```

```
# 문자 출력
extern printd(x);
extern putchard(char);
# 명령문을 연결
def binary : 1 (x y) 0;
# Logical unary not.
def unary!(v)
  if v then    0  else    1;
# -value : 단항 연산자
def unary-(v)  0-v;
```

47) https://en.wikipedia.org/wiki/Mandelbrot_set

```
# Define > with the same precedence as <.
# def binary> 10 (LHS RHS)  RHS < LHS;

# Binary "logical or", (note that it does not "short circuit")
def binary| 5 (LHS RHS)
  if LHS then    1  else if RHS then    1  else    0;

# Define = with slightly lower precedence than relationals.
def binary= 9 (LHS RHS)  !(LHS < RHS | LHS > RHS);
```

```
# mandelbrot 관련 함수
def printdensity(d)
  if d > 8 then  putchard(32)
  else if d > 4 then  putchard(46)
  else if d > 2 then  putchard(43)
  else   putchard(42);

def mandelconverger(real imag iters creal cimag)
  if iters > 255 | (real*real + imag*imag > 4) then   iters
  else   mandelconverger(real*real - imag*imag + creal,
                    2*real*imag + cimag, iters+1, creal, cimag);

def mandelconverge(real imag)
  mandelconverger(real, imag, 0, real, imag);

def mandelhelp(xmin xmax xstep   ymin ymax ystep)
  for y = ymin, y < ymax, ystep in (
    (for x = xmin, x < xmax, xstep in
       printdensity(mandelconverge(x,y))) : putchard(10)
  );

def mandel(realstart imagstart realmag imagmag)
  mandelhelp(realstart, realstart+realmag*78, realmag,
             imagstart, imagstart+imagmag*40, imagmag);
```

```
ready> mandel(-2.3, -1.3, 0.05, 0.07);
ready> ********************************************************************************
********************************************************************************
********************************************************************************
**********************************++++++++...++++++++++++++++++++++++++++++++++++++++
**********************************+++++++.. ...++++++++++++++++++++++++++++++++++++++
*********************************++++++..   ..++++++++++++++++++++++++++++++++++++++
*********************************+++++.     ..++++++++++++++++++++++++++++++++++++++
*********************************++++....     ..++++++++++++++++++++++++++++++++++++++
*********************************++++.......     .....++++++++++++++++++++++++++++++
*********************************+++.   .          ... .++++++++++++++++++++++++++++
*********************************++++...                +++++++++++++++++++++++++++
*********************************++++....                .+++++++++++++++++++++++++++
*********************************++..++....                ..+++++++++++++++++++++++++++
*********************************. ..........                 +++++++++++++++++++++++++++
*********************************..       ..                  .+++++++++++++++++++++++++++
*********************************...                         .+++++++++++++++++++++++++++
*********************************..                          .+++++++++++++++++++++++++++
*********************************.....                       ..+++++++++++++++++++++++++++
*********************************+.....                      ...+++++++++++++++++++++++++++
*********************************+... ....                   ...+++++++++++++++++++++++++++
*********************************+......                     ..+++++++++++++++++++++++++++
*********************************...                         .+++++++++++++++++++++++++++
*********************************...                          +++++++++++++++++++++++++++
*********************************..       ..                  ..++++++++++++++++++++++++++
*********************************.. ..........                 +++++++++++++++++++++++++++
*********************************...+++.....                  ..++++++++++++++++++++++++++
*********************************++++....                    ..++++++++++++++++++++++++++
*********************************+++...                       +++++++++++++++++++++++++++
*********************************..   .          ... .++++++++++++++++++++++++++++
*********************************+++++.....     ......++++++++++++++++++++++++++++++
*********************************++++....      ..++++++++++++++++++++++++++++++++++
*********************************..     ..++++++++++++++++++++++++++++++++++++++++
*********************************..  ...++++++++++++++++++++++++++++++++++++++++++
*********************************.. ...++++++++++++++++++++++++++++++++++++++++++
```

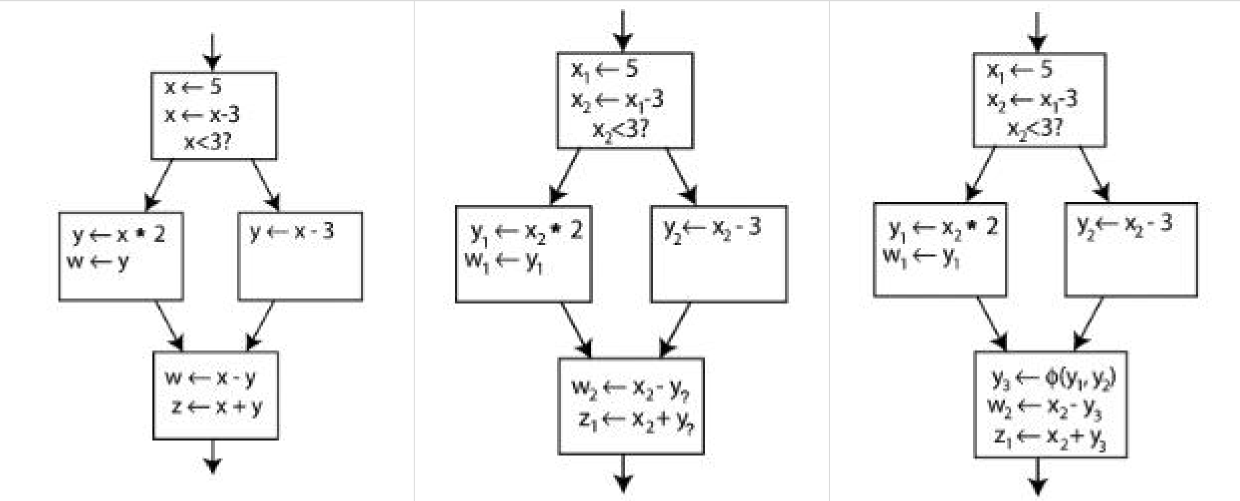
```
************************************++++++....+++++*****************************
****************************************++++++++********************************
********************************************************************************
********************************************************************************
********************************************************************************
********************************************************************************
********************************************************************************
Evaluated to 0.000000
```

3.7. 확장하기: 변경가능한 변수

- 함수형 언어는 LLVM IR을 SSA 형태로 직접 빌드하는 것을 매우 쉽게 만든다.

※ SSA(static single assignment form, 정적 단일 할당 형식)48)는 1988년 IBM에 근무하였던 Barry K. Rosen, Mark N. Wegman, F. Kenneth Zadeck에 의해 제안. 49) 다음과 같은 특징으로 인하여 컴파일러에서 사용되는 중간표현으로 널리 사용된다.

- SSA의 특징: ① 각 변수는 정확히 한 번만 할당됨. 모든 변수 사용은 해당 변수의 단일 정의를 참조함. 변수에 새 값을 할당할 때마다 새로운 버전의 변수가 생성됨. ② φ(phi) 함수는 제어의 흐름이 합쳐지는 지점에서 여러 정의 중 어떤 것을 사용할지 결정하는데 사용됨. ③ 변수의 정의와 사용 관계가 명확해져 데이터 흐름 분석이 더 쉬워짐 ④ 상수 전파, 사용되지 않는 코드 제거 등의 최적화가 더 효과적으로 수행될 수 있음 ⑤ 메모리 접근을 최소화 하고, 레지스터 할당이 더 효율적으로 이루어질 수 있음 ⑥ 변수의 추적이 용이하여 디버깅이 쉬워진다.



일반적인 CFG 그래프	SSA 형식으로 변환. $y_?$ CFG 흐름에 따라 결정됨	φ(y_1,y_2) 노드를 추가함 y_3 ← y_1 또는 y_2

- 아래 예시에서 가변 변수가 SSA 구성에서 복잡성을 유발하는 이유: 변수 X는 프로그램의 실행 경로에 따라 달라짐. 리턴전에 X는 두 가지 가능한 값이 있기 때문에 phi 노드를 삽입하여 두 값을 병합한다.

```
int G, H;
int test(_Bool Condition) {
  int X;
  if (Condition)   X = G;
  else   X = H;
  return X;
}
```

```
@G = weak global i32 0   ; type of @G is i32*
@H = weak global i32 0   ; type of @H is i32*
```

48) https://en.wikipedia.org/wiki/Static_single-assignment_form
49) https://dl.acm.org/doi/10.1145/75277.75280

```
define i32 @test(i1 %Condition) {
entry:
    분기 branch 50)
  br i1 %Condition, label %cond_true, label %cond_false

cond_true:
  %X.0 = load i32, i32* @G
  br label %cond_next

cond_false:
  %X.1 = load i32, i32* @H
  br label %cond_next

cond_next:
 phi 명령은 SSA 형식을 구현하는 역할 51)
   제어 흐름이 cond_false 에서 오는 경우 X.2는 X.1의 값을 가져옴.
   제어 흐름이 cond_true 에서 오는 경우 X.2 는 X.0의 값을 가져옴
  %X.2 = phi i32 [ %X.1, %cond_false ], [ %X.0, %cond_true ]
  ret i32 %X.2
}
```

- LLVM에서 모든 메모리 엑세스는 load/store 명령어로 명시적으로만 접근이 가능하며 “address-of" 연산자가 없도록(또는 필요하지 않도록) 설계됨. @G/@H 전역변수의 유형이 "i32"로 정의되었음에도 불구하고 실제로는 "i32*"임에 주의한다. 스택 변수도 이와 같은 방식으로 사용된다. 스택 변수는 alloca[52] 명령으로 선언된다. 위의 예제 코드를 phi 노드를 사용하지 않고, alloca 명령을 이용하여 재작성하면 다음과 같다.

```
; tut7-test.ll
@G = weak global i32 0   ; type of @G is i32*
@H = weak global i32 0   ; type of @H is i32*

define i32 @test(i1 %Condition) {
entry:
  %X = alloca i32           ; type of %X is i32*.
  br i1 %Condition, label %cond_true, label %cond_false

cond_true:
  %X.0 = load i32, i32* @G
  store i32 %X.0, i32* %X   ; Update X
  br label %cond_next

cond_false:
  %X.1 = load i32, i32* @H
  store i32 %X.1, i32* %X   ; Update X
  br label %cond_next

cond_next:
  %X.2 = load i32, i32* %X  ; Read X
  ret i32 %X.2
}
```

50) https://llvm.org/docs/LangRef.html#br-instruction
51) https://llvm.org/docs/LangRef.html#phi-instruction
52) https://llvm.org/docs/LangRef.html#alloca-instruction

- 임의의 가변 변수는 스택에 저장되며, 스택에서 읽을 때는 load, 스택을 업데이트할 때는 store가 작동한다. 일반적인 작업에 스택(메모리) 트래픽이 필요하므로 성능 문제를 일으킬 수 있지만 LLVM의 최적화 프로그램에는 "mem2reg"라는 최적화 패스가 있어, 메모리 접근을 레지스터 연산으로 승격시키고, 적절한 경우 phi 노드를 삽입한다. 이 패스를 통해 실행하면 다음과 같은 결과를 얻는다.

```
    mem2reg 패스를 통하여 위 IR코드를 최적화함
$ opt  -passes=mem2reg  -f tut7-test.ll  ¦ llvm-dis
; ModuleID = '<stdin>'
source_filename = "tut7-test.ll"
@G = weak global i32 0
@H = weak global i32 0
    ; mem2reg 최적화에 의해 스택 변수와 store 문이 사라지고, phi 노드가 추가됨
define i32 @test(i1 %Condition) {
entry:
  br i1 %Condition, label %cond_true, label %cond_false

cond_true:                                        ; preds = %entry
  %X.0 = load i32, ptr @G, align 4
  br label %cond_next

cond_false:                                       ; preds = %entry
  %X.1 = load i32, ptr @H, align 4
  br label %cond_next

cond_next:                                        ; preds = %cond_false, %cond_true
  %X.01 = phi i32 [ %X.0, %cond_true ], [ %X.1, %cond_false ]
  ret i32 %X.01
}
```

- mem2reg 패스는 SSA 폼을 구성하기 위한 지배자 프런티어(dominator frontier) 알고리즘[53]을 사용하여 PHI노드를 적절한 위치에 배치함. 작동 원리는 대략 다음과 같다 : ① alloca 를 찾고 처리할 수 있으면 alloca를 승격시킴. 전역 변수나 힙 할당에는 적용되지 않는다. ② 함수의 엔트리 블록에서만 alloca 명령을 찾음 ③ 직접 로드 및 저장에 사용되는 alloca 만 승격한다. 스택 객체의 주소가 함수에 전달되거나 포인터 산술이 관련된 경우 alloca는 관여하지 않음. ④ first-class 값[54](포인터, 스칼라, 벡터 등)에 대해서만 작동하며, 할당의 배열 크기가 1인 경우에만 작동함. 구조체나 배열을 레지스터로 승격할 수 없음. SROA(scalar replacement of aggregates) 패스가 더 강력하고 많은 경우 구조체, 유니언 배열을 승격할 수 있음.

- Kal 언어에서 가변 변수 문제는 다음의 두 가지 기능을 추가하는 것이다:
  ① '=' 연산자로 변수를 변형할 수 있는 기능 ② 새로운 변수를 정의하는 능력
- 다음은 이러한 변수를 사용하는 방법을 보여주는 예시임.

```
# fibi.kal
# 낮은 우선순위의 ':' 시퀀스 연결 연산자. 항상 RHS를 리턴
def binary : 1 (x y) y;

# 재귀호출을 이용한 피보나치 수 계산
def fib(x)
```

53) https://en.wikipedia.org/wiki/Dominator_%28graph_theory%29
54) https://llvm.org/docs/LangRef.html#t-firstclass

```
  if (x < 3) then    1
  else    fib(x-1)+fib(x-2);

# Iterative fib.
def fibi(x)
  var a = 1, b = 1, c in
  (for i = 3, i < x in
     c = a + b :     a = b :     b = c) :  b;

# Call it.
fibi(10);
```

- 심볼 테이블은 코드 생성 시간에 NamedValues 맵에 의해 관리됨. 이 맵은 명명된 변수에 대한 double 값을 보관하는 LLVM "Value *"를 추적함.

- 변수를 변경하려면 기존 변수를 변경하여 "alloca"를 사용해야 함. Value* 대신 AllocaInst* 에 매핑되도록 맵을 변경. AllocaInst는 Value 클래스를 간접적으로 상속받는다 (상속 계층 : Value -> User -> Instruction -> UnaryInstruction -> AllocaInst)

```
static std::map<std::string, AllocaInst *> NamedValues;
```

- alloca를 생성하기 위해 entry 블록에 alloca가 생성되도록 도우미 함수를 사용

```
/// CreateEntryBlockAlloca - Create an alloca instruction in the entry block of
/// the function.  This is used for mutable variables etc.
static AllocaInst *CreateEntryBlockAlloca(Function *TheFunction,
                                          const std::string &VarName) {
  // 엔트리 블록의 첫번째 명령어(.begin())를 가르키는 IRBuilder 객체를 생성하고 alloca 명령을
생성하여 반환
  IRBuilder<> TmpB(&TheFunction->getEntryBlock(),
                   TheFunction->getEntryBlock().begin());
  return TmpB.CreateAlloca(Type::getDoubleTy(*TheContext), nullptr, VarName);
}
```

- 변수를 참조. 변수는 스택에 존재하므로 로드 명령을 생성한다.

```
Value *VariableExprAST::codegen() {
  // Look this variable up in the function.
  AllocaInst *A = NamedValues[Name];
  if (!A)
    return LogErrorV("Unknown variable name");

  // load 명령을 생성
  return Builder->CreateLoad(A->getAllocatedType(), A, Name.c_str());
}
```

- for 루프 AST 클래스 수정

```
Value *ForExprAST::codegen() {
  Function *TheFunction = Builder->GetInsertBlock()->getParent();

  // 루프 변수에 대한 alloca 명령을 생성. 위에서 만든 도우미 함수를 이용
  AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, VarName);
```

```
  // Emit the start code first, without 'variable' in scope.
  Value *StartVal = Start->codegen();
  if (!StartVal)    return nullptr;

  // 시작 값을 저장하는 명령
  Builder->CreateStore(StartVal, Alloca);

  BasicBlock *LoopBB = BasicBlock::Create(*TheContext, "loop", TheFunction);
  Builder->CreateBr(LoopBB);
  Builder->SetInsertPoint(LoopBB);

  // 기존 변수를 섀도잉하기 위해 보관
  AllocaInst *OldVal = NamedValues[VarName];
  NamedValues[VarName] = Alloca;
[...]

  // 루프 변수를 re-load, increment, store 함. (함수의 body에서 루프 변수를 변경할 수 있음)
  Value *CurVar = Builder->CreateLoad(Alloca->getAllocatedType(), Alloca, VarName.c_str());
  Value *NextVar = Builder->CreateFAdd(CurVar, StepVal, "nextvar");
  Builder->CreateStore(NextVar, Alloca);
[...]
```

- 함수 AST 클래스 수정

```
Function *FunctionAST::codegen() {
  ...
  Builder->SetInsertPoint(BB);

  // Record the function arguments in the NamedValues map.
  NamedValues.clear();
  for (auto &Arg : TheFunction->args()) {
    // 함수 매개변수들에 대해서 alloca 를 생성함
    AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, Arg.getName());

    // 초기 값을 저장하는 명령
    Builder->CreateStore(&Arg, Alloca);

    // 변수 이름을 심볼테이블에 저장
    NamedValues[std::string(Arg.getName())] = Alloca;
  }

  if (Value *RetVal = Body->codegen()) {
    ...
```

- 마지막으로 mem2reg 패스를 추가한다.

```
// Promote allocas to registers.
TheFPM->add(createPromoteMemoryToRegisterPass());
// Do simple "peephole" optimizations and bit-twiddling optzns.
TheFPM->add(createInstructionCombiningPass());
// Reassociate expressions.
TheFPM->add(createReassociatePass());
...
```

- 새로운 할당 연산자 '='를 추가함

```
int main() {
  // Install standard binary operators.
  // 1 is lowest precedence.
  BinopPrecedence['='] = 2;
  BinopPrecedence['<'] = 10;
  BinopPrecedence['+'] = 20;
  BinopPrecedence['-'] = 20;
```

- 이진연산자 AST 클래스 수정

```
Value *BinaryExprAST::codegen() {
  // Special case '=' because we don't want to emit the LHS as an expression.
  if (Op == '=') {
    VariableExprAST *LHSE = static_cast<VariableExprAST*>(LHS.get());
    if (!LHSE)      return LogErrorV("destination of '=' must be a variable");
    // Codegen the RHS.
    Value *Val = RHS->codegen();
    if (!Val)    return nullptr;
    // Look up the name.
    Value *Variable = NamedValues[LHSE->getName()];
    if (!Variable)    return LogErrorV("Unknown variable name");

    Builder->CreateStore(Val, Variable);
    return Val;
  }
...
```

- 할당 연산자(=) 테스트.

```
# 테스트용 Kal 언어 코드
extern printd(x);
def binary : 1 (x y) y;
def test(x)
  printd(x) :  x = 4 :  printd(x);
test(123);
아래 변환 패스들을 잠시 주석처리하고 테스트한 결과는 아래와 같다.
  TheFPM->addPass(InstCombinePass());
  TheFPM->addPass(ReassociatePass());
  TheFPM->addPass(GVNPass());
  TheFPM->addPass(SimplifyCFGPass());
[...]
Read function definition:define double @test(double %x) {
entry:
  %x1 = alloca double, align 8
  store double %x, ptr %x1, align 8
  %x2 = load double, ptr %x1, align 8
  %calltmp = call double @printd(double %x2)   <---  printd(x)
  store double 4.000000e+00, ptr %x1, align 8   <--- x = 4
  %binop = call double @"binary:"(double %calltmp, double 4.000000e+00) <--- binary:
  %x3 = load double, ptr %x1, align 8
  %calltmp4 = call double @printd(double %x3)    <--- printd(x)
  %binop5 = call double @"binary:"(double %binop, double %calltmp4)  <--- binary:
  ret double %binop5
}
ready> test(123);
```

```
ready> 123.000000
4.000000   <------ 함수 안에서 4.0 로 값이 변경된 다음 출력됨
Evaluated to 0.000000
   변환 패스를 활성화 했을때 IR 결과
Read function definition:define double @test(double %x) {
entry:
  %calltmp = call double @printd(double %x)   <--- printd(x) : 123.0을 출력
  %binop = call double @"binary:"(double %calltmp, double 4.000000e+00)  <--- binary:
  %calltmp4 = call double @printd(double 4.000000e+00)   <--- printd(x) 4.0을 출력
  %binop5 = call double @"binary:"(double %binop, double %calltmp4)  <--- binary:
  ret double %binop5
}
```

- 다음으로 사용자 정의 지역 변수 기능을 확장하기. var/in 토큰을 추가한다.

```
enum Token {
  tok_for = -9,
  tok_in = -10,
  // var definition
  tok_var = -13
static int gettok() {
[...]
    if (IdentifierStr == "var")
      return tok_var;
```

- 변수 AST 클래스 작성

```
/// VarExprAST - Expression class for var/in
class VarExprAST : public ExprAST {
  std::vector<std::pair<std::string, std::unique_ptr<ExprAST>>> VarNames;  // 변수명
  std::unique_ptr<ExprAST> Body;

public:
  VarExprAST(std::vector<std::pair<std::string, std::unique_ptr<ExprAST>>> VarNames,
             std::unique_ptr<ExprAST> Body)
    : VarNames(std::move(VarNames)), Body(std::move(Body)) {}

  Value *codegen() override;
};
```

- 주표현식에 추가하기

```
static std::unique_ptr<ExprAST> ParsePrimary() {
[...]
  case tok_var:
    return ParseVarExpr();
  }
}
```

- ParseVarExpr 을 정의

```
/// varexpr ::= 'var' identifier ('=' expression)?
//                    (',' identifier ('=' expression)?)* 'in' expression
static std::unique_ptr<ExprAST> ParseVarExpr() {
  getNextToken();  // eat the var.
```

```
  std::vector<std::pair<std::string, std::unique_ptr<ExprAST>>> VarNames;

  // At least one variable name is required.
  if (CurTok != tok_identifier)  return LogError("expected identifier after var");

  while (true) {
    std::string Name = IdentifierStr;
    getNextToken();  // eat identifier.

    // Read the optional initializer.
    std::unique_ptr<ExprAST> Init;
    if (CurTok == '=') {
      getNextToken(); // eat the '='.
      Init = ParseExpression();
      if (!Init) return nullptr;
    }

    VarNames.push_back(std::make_pair(Name, std::move(Init)));

    // End of var list, exit loop.
    if (CurTok != ',') break;
    getNextToken(); // eat the ','.

    if (CurTok != tok_identifier) return LogError("expected identifier list after var");
  }

  // At this point, we have to have 'in'.
  if (CurTok != tok_in) return LogError("expected 'in' keyword after 'var'");
  getNextToken();  // eat 'in'.

  auto Body = ParseExpression();
  if (!Body)  return nullptr;

  return std::make_unique<VarExprAST>(std::move(VarNames), std::move(Body));
}
```

- IR 코드 생성

```
Value *VarExprAST::codegen() {
  std::vector<AllocaInst *> OldBindings;

  Function *TheFunction = Builder->GetInsertBlock()->getParent();

  // Register all variables and emit their initializer.
  for (unsigned i = 0, e = VarNames.size(); i != e; ++i) {
    const std::string &VarName = VarNames[i].first;
    ExprAST *Init = VarNames[i].second.get();

    // Emit the initializer before adding the variable to scope, this prevents
    // the initializer from referencing the variable itself, and permits stuff
    // like this:
    //  var a = 1 in
    //    var a = a in ...   # refers to outer 'a'.
    Value *InitVal;
    if (Init) {
```

```
      InitVal = Init->codegen();
      if (!InitVal)
        return nullptr;
    } else { // If not specified, use 0.0.
      InitVal = ConstantFP::get(*TheContext, APFloat(0.0));
    }

    AllocaInst *Alloca = CreateEntryBlockAlloca(TheFunction, VarName);
    Builder->CreateStore(InitVal, Alloca);

    // Remember the old variable binding so that we can restore the binding when
    // we unrecurse.
    OldBindings.push_back(NamedValues[VarName]);

    // Remember this binding.
    NamedValues[VarName] = Alloca;
  }

  // Codegen the body, now that all vars are in scope.
  Value *BodyVal = Body->codegen();
  if (!BodyVal)
    return nullptr;

  // Pop all our variables from scope.
  for (unsigned i = 0, e = VarNames.size(); i != e; ++i)
    NamedValues[VarNames[i].first] = OldBindings[i];

  // Return the body computation.
  return BodyVal;
}
```

- 테스트

```
$ clang++ -g -O3 `llvm-config --cxxflags --ldflags --system-libs --libs core orcjit native`\
   -lstar -L.  tut7.cpp -o tut7 ; export LD_LIBRARY_PATH=. ; ./tut7
# var / in 표현식 테스트 / fibi() 함수 안에서 a,b,c 변수를 정의함
# for loop 이 종료된 이후 최종 결과는 b에 저장됨
def binary : 1 (x y) y;
def fibi(x)
  var a = 1, b = 1, c in
  (for i = 3, i < x in
     c = a + b :     a = b :     b = c) :  b;
fibi(10);
   최적화 관련 변환 패스들을 잠시 주석처리하고 테스트한 결과
Read function definition:define double @fibi(double %x) {
entry:
  %i = alloca double, align 8
  %c = alloca double, align 8
  %b = alloca double, align 8
  %a = alloca double, align 8
  %x1 = alloca double, align 8
  store double %x, ptr %x1, align 8
  store double 1.000000e+00, ptr %a, align 8
  store double 1.000000e+00, ptr %b, align 8
```

```
  store double 0.000000e+00, ptr %c, align 8
  store double 3.000000e+00, ptr %i, align 8
  br label %loop

loop:                                             ; preds = %loop, %entry
  %a2 = load double, ptr %a, align 8    <--- load 1
  %b3 = load double, ptr %b, align 8    <--- load 2
  %addtmp = fadd double %a2, %b3
  store double %addtmp, ptr %c, align 8    <--- store 1
  %b4 = load double, ptr %b, align 8    <--- load 3
  store double %b4, ptr %a, align 8    <--- store 2
  %binop = call double @"binary:"(double %addtmp, double %b4)
  %c5 = load double, ptr %c, align 8    <--- load 4
  store double %c5, ptr %b, align 8    <--- store 3
  %binop6 = call double @"binary:"(double %binop, double %c5)
  %i7 = load double, ptr %i, align 8    <--- load 5
  %x8 = load double, ptr %x1, align 8    <--- load 6
  %cmptmp = fcmp ult double %i7, %x8
  %booltmp = uitofp i1 %cmptmp to double
  %i9 = load double, ptr %i, align 8    <--- load 7
  %nextvar = fadd double %i9, 1.000000e+00
  store double %nextvar, ptr %i, align 8    <--- store 4
  %loopcond = fcmp one double %booltmp, 0.000000e+00
  br i1 %loopcond, label %loop, label %afterloop
          loop 내부에서 총 7번의 load, 4번의 store 가 실행
afterloop:                                        ; preds = %loop
  %b10 = load double, ptr %b, align 8    <--- load 8
  %binop11 = call double @"binary:"(double 0.000000e+00, double %b10)
  ret double %binop11
}

ready> Evaluated to 55.000000   <----  1,1,2,3,5,8,13,21,34,55=fibi(10)
   변환 패스를 활성화 했을때 IR 결과
Read function definition:define double @fibi(double %x) {
entry:
  %i = alloca double, align 8
  %b = alloca double, align 8
  %a = alloca double, align 8
  %x1 = alloca double, align 8
  store double %x, ptr %x1, align 8
  store double 1.000000e+00, ptr %a, align 8
  store double 1.000000e+00, ptr %b, align 8
  store double 3.000000e+00, ptr %i, align 8
  br label %loop

loop:                                             ; preds = %loop, %entry
  %i7 = phi double [ %nextvar, %loop ], [ 3.000000e+00, %entry ]
  %b3 = phi double [ %addtmp, %loop ], [ 1.000000e+00, %entry ]
  %a2 = phi double [ %b3, %loop ], [ 1.000000e+00, %entry ]
  %addtmp = fadd double %a2, %b3
  store double %b3, ptr %a, align 8     <--- store 1
  %binop = call double @"binary:"(double %addtmp, double %b3)
  store double %addtmp, ptr %b, align 8     <--- store 2
  %binop6 = call double @"binary:"(double %binop, double %addtmp)
```

```
  %cmptmp = fcmp ult double %i7, %x
  %nextvar = fadd double %i7, 1.000000e+00
  store double %nextvar, ptr %i, align 8     <--- store 3
  br i1 %cmptmp, label %loop, label %afterloop
          loop 내부에서 총 3번의 store 가 실행
afterloop:                                        ; preds = %loop
  %binop11 = call double @"binary:"(double 0.000000e+00, double %addtmp)
  ret double %binop11
}
```

3.8. 오브젝트 코드로 컴파일

- LLVM은 크로스컴파일을 기본적으로 지원함. 타겟으로 삼고 싶은 아키텍처를 지정하기 위해 "target tripple" 이라는 문자열을 사용. clang에서 현재 시스템의 타겟을 조회하기

```
$ clang --version ¦ grep Target
Target: x86_64-pc-linux-gnu
   ARM 플롯폼에서
$ clang --version ¦ grep Target
Target: armv6k-unknown-linux-gnueabihf
```

- 다행히도 현재 머신을 타겟으로 하기 위해 타겟 트리플을 하드코딩 할 필요는 없음. LLVM은 현재 머신의 타겟을 조회할 수 있다.

```
auto TargetTriple = sys::getDefaultTargetTriple();
```

- 오브젝트 코드를 출력할 타겟을 초기화 하기

```
InitializeAllTargetInfos();   // 모든 사용가능한 타겟 정보를 초기화
InitializeAllTargets();
InitializeAllTargetMCs();
InitializeAllAsmParsers();
InitializeAllAsmPrinters();
```

- 타겟 트리플을 사용하여 해당 타겟을 조회함.

```
std::string Error;
auto Target = TargetRegistry::lookupTarget(TargetTriple, Error);
if (!Target) {
  errs() << Error;  return 1;
}
```

- TargetMachine 이라는 클래스가 필요한데. 이것은 타겟으로 하는 머신의 정보를 담고 있다. CPU의 특정 기능(SSE와 같은)을 포함하고 싶다면 Features에 포함하면 된다.
- CPU 지원되는 features를 조회하기. llc를 이용

```
$ llc -march=x86 -mattr=help
[...]
Available features for this target:
  16bit-mode                     - 16-bit mode (i8086).
  32bit-mode                     - 32-bit mode (80386).
[...]
```

- 추가 기능이나 대상 옵션 없이 일반 CPU를 사용

```
auto CPU = "generic";
auto Features = "";

TargetOptions opt;
auto TargetMachine
   = Target->createTargetMachine(TargetTriple, CPU, Features, opt, Reloc::PIC_);
```

- 모듈에서 데이터 레이아웃과 타겟을 지정하기

```
TheModule->setDataLayout(TargetMachine->createDataLayout());
TheModule->setTargetTriple(TargetTriple);
```

- 객체 코드를 파일로 내보낼 준비

```
auto Filename = "output.o";  // 내보내기할 파일명
std::error_code EC;
raw_fd_ostream dest(Filename, EC, sys::fs::OF_None);
if (EC) {
  errs() << "Could not open file: " << EC.message();  return 1;
}
```

- 개체 코드를 내보내는 패스를 정의하고 실행

```
legacy::PassManager pass;
auto FileType = CodeGenFileType::ObjectFile; // 오브젝트 파일로 내보낼 예정

if (TargetMachine->addPassesToEmitFile(pass, dest, nullptr, FileType)) {
  errs() << "TargetMachine can't emit a file of this type";
  return 1;
}
pass.run(*TheModule);
dest.flush();
```

- 오브젝트 파일 내보내기 테스트

```
  평균을 구하는 함수
$ more average.kal
def average(x y) (x + y) * 0.5;
```

```
// average_main.c
#include <iostream>
extern "C" {
  double average(double, double);
}
int main() {
  std::cout << "average of 3.0 and 4.0: " << average(3.0, 4.0) << std::endl;
}
```

```
$ clang++ -g -O3 `llvm-config --cxxflags --ldflags --system-libs --libs core orcjit native` \
   tut8.cpp -o tut8
$ ./tut8 < average.kal
[...]
Wrote output.o
$ file output.o
output.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), not stripped
```

```
$ clang++  average_main.cpp  output.o   -o average_main
$ ./average_main
average of 3.0 and 4.0: 3.5
$ gdb ./average_main
(gdb) disas average          gdb로 average 함수를 역어셈블 한 결과
Dump of assembler code for function average:
   0x00000000000011c0 <+0>:     movsd  %xmm0,-0x10(%rsp)   매개변수 x
   0x00000000000011c6 <+6>:     movsd  %xmm1,-0x8(%rsp)    매개변수 y
   0x00000000000011cc <+12>:    addsd  %xmm1,%xmm0         x + y
   0x00000000000011d0 <+16>:    mulsd  0xe40(%rip),%xmm0        # 0x2018
   0x00000000000011d8 <+24>:    ret
End of assembler dump.
(gdb) print/x 0x00000000000011d8+0xe40                  상수값이 저장된 메모리 주소
$2 = 0x2018
(gdb) x/gf  0x2018                                      상수값은 0.5
0x2018: 0.5
```

3.9. 소스레벨 디버깅

- 디버거에서 활용할 수 있는 디버그 정보는 DWARF[55] 라는 포맷을 사용함. 디버깅 정보는 유형, 소스 위치, 변수 위치 등에 관한 정보를 인코딩해서 담고 있음. 최적화 과정은 소스 위치를 추적하는 것을 어렵게 만들 수 있다. LLVM IR의 각 명령은 원해 소스 위치를 유지함. 명령어가 병합될 경우 단위 위치만 유지하게 되므로 프로그램을 단계적으로 실행할 때 점프가 발생할 수 있다. 디버깅을 위해서는 최적화를 피하는 것이 좋을 수 있음.

- JIT를 통해 디버깅할 수 없으므로 JIT 기능은 제거하고, 컴파일 기능만 유지한다. 먼저 최상위 수준 문장을 포함하는 익명 함수를 "main"으로 만들어 준다.

```
/// toplevelexpr ::= expression
static std::unique_ptr<FunctionAST> ParseTopLevelExpr() {
  SourceLocation FnLoc = CurLoc;
  if (auto E = ParseExpression()) {
    // Make the top-level expression be our "main" function.
    auto Proto = std::make_unique<PrototypeAST>(FnLoc, "main", std::vector<std::string>());
    return std::make_unique<FunctionAST>(std::move(Proto), std::move(E));
  }
  return nullptr;
}
```

- JIT 프롬프트를 출력하는 부분을 제거한다.

```
// fprintf(stderr, "ready> ");
```

- 모든 최적화 패스와 JIT 를 비활성화 한다.

```
// TheJIT = ExitOnErr(KaleidoscopeJIT::Create());
[...]
static void HandleTopLevelExpression() {
  // Evaluate a top-level expression into an anonymous function.
  if (auto FnAST = ParseTopLevelExpr()) {
    if (FnAST->codegen()) {
```

55) https://dwarfstd.org/

```
      fprintf(stderr, "Error generating code for top level expr");
      // Create a ResourceTracker to track JIT'd memory allocated to our
      // anonymous expression -- that way we can free it after executing.
//      auto RT = TheJIT->getMainJITDylib().createResourceTracker();

//      auto TSM = ThreadSafeModule(std::move(TheModule), std::move(TheContext));
//      ExitOnErr(TheJIT->addModule(std::move(TSM), RT));
//      InitializeModuleAndManagers();

      // Search the JIT for the __anon_expr symbol.
//      auto ExprSymbol = ExitOnErr(TheJIT->lookup("__anon_expr"));

      // Get the symbol's address and cast it to the right type (takes no
      // arguments, returns a double) so we can call it as a native function.
//      double (*FP)() = ExprSymbol.getAddress().toPtr<double (*)()>();
//      fprintf(stderr, "Evaluated to %f\n", FP());

      // Delete the anonymous expression module from the JIT.
//      ExitOnErr(RT->remove());
    }
[...]
    // Run the optimizer on the function.
    // TheFPM->run(*TheFunction, *TheFAM);
```

- DWARF의 코드 섹션에 대한 최상위 컨테이너는 컴파일 단위임. 여기에는 타입 정보와 함수 데이터가 들어있다. IRBuider와 유사하게 DBuilder 클래스는 IR 파일에서 디버깅 메타데이터를 구성하는데 도움을 준다. 자세한 것은 소스레벨 디버깅 문서[56]를 참조한다.

```
static std::unique_ptr<DIBuilder> DBuilder;

struct DebugInfo {
  DICompileUnit *TheCU;
  DIType *DblTy;
  DIType *getDoubleTy();
} KSDbgInfo;

DIType *DebugInfo::getDoubleTy() {
  if (DblTy)    return DblTy;
  DblTy = DBuilder->createBasicType("double", 64, dwarf::DW_ATE_float);
  return DblTy;
}

void DebugInfo::emitLocation(ExprAST *AST) {
  if (!AST)    return Builder->SetCurrentDebugLocation(DebugLoc());
  DIScope *Scope;
  if (LexicalBlocks.empty())    Scope = TheCU;
  else    Scope = LexicalBlocks.back();
  Builder->SetCurrentDebugLocation(DILocation::get(
      Scope->getContext(), AST->getLine(), AST->getCol(), Scope));
}

static DISubroutineType *CreateFunctionType(unsigned NumArgs) {
```

56) https://llvm.org/docs/SourceLevelDebugging.html

```
  SmallVector<Metadata *, 8> EltTys;
  DIType *DblTy = KSDbgInfo.getDoubleTy();
  // Add the result type.
  EltTys.push_back(DblTy);
  for (unsigned i = 0, e = NumArgs; i != e; ++i)
    EltTys.push_back(DblTy);
  return DBuilder->createSubroutineType(DBuilder->getOrCreateTypeArray(EltTys));
}
```

- Kaleidoscope라는 언어에 대한 컴파일 단위를 생성하는 동안 C언어의 ABI를 따르도록 함으로써 실제로 디버거에서 함수를 호출하거나 직접적으로 실행할 수 있게 한다. 소스코드는 표준입력으로 부터 리다이렉트 되지만 createCompileUnit 에서 "fib.ks"를 하드코드 함으로써 소스파일의 파일명을 지정하여 준다.

```
  // Construct the DIBuilder, we do this here because we need the module.
  DBuilder = std::make_unique<DIBuilder>(*TheModule);

  // Create the compile unit for the module.
  // Currently down as "fib.ks" as a filename since we're redirecting stdin
  // but we'd like actual source locations.
  KSDbgInfo.TheCU = DBuilder->createCompileUnit(
      dwarf::DW_LANG_C, DBuilder->createFile("fib.ks", "."),
      "Kaleidoscope Compiler", false, "", 0);
[...]
  DBuilder->finalize();
```

- 테스트하기

```
$ more fib.ks
def fib(x)
  if x < 3 then 1
  else fib(x-1)+fib(x-2);

fib(10);
$ clang++ -g -O3 `llvm-config --cxxflags --ldflags --system-libs --libs core orcjit native` \
   tut9.cpp -o tut9

$ ./tut9 < fib.ks |& clang -x ir -o fib -
$ file fib
fib: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /
lib64/ld-linux-x86-64.so.2, BuildID[sha1]=d1c5e12c4df8fedc7bd1b53a487369279c181521, for GNU/Lin
ux 3.2.0, with debug_info, not stripped

$ objdump -j .text --disassemble=fib  fib         fib 함수의 역어셈블 결과
fib:     file format elf64-x86-64
Disassembly of section .text:
0000000000001130 <fib>:
    1130:       48 83 ec 18             sub    $0x18,%rsp
    1134:       f2 0f 11 44 24 10       movsd  %xmm0,0x10(%rsp)
    113a:       f2 0f 10 4c 24 10       movsd  0x10(%rsp),%xmm1
    1140:       f2 0f 10 05 c0 0e 00    movsd  0xec0(%rip),%xmm0        # 2008 <_IO_stdin_used+0x8>
    1147:       00
[...]
    11c8:       c3                      ret
```

[참고자료]

- LLVM을 이용한 컴파일 및 IR파일 읽기 https://roadtosuccess.tistory.com/39
- LLVM Language Reference Manual https://llvm.org/docs/LangRef.html
- LLVM Programmer's Manual https://llvm.org/docs/ProgrammersManual.html
- LLVM Command Guide https://llvm.org/docs/CommandGuide/
- Introduction to LLVM, Eric Christopher, Johannes Doerfert https://www.youtube.com/watch?v=J5xExRGaIIY
- LLVM Tutorial https://llvm.org/docs/tutorial/
- Clang Tutorial Part 1,2,3: https://kevinaboos.wordpress.com/2013/07/23/clang-tutorial-part-i-introduction/
- CLang tutorial to build a branch coverage measuring tool, KAIST CS453 https://swtv.kaist.ac.kr/courses/cs453-fall13/
- LLVM Tutorial, University of Rochester https://www.cs.rochester.edu/u/criswell/asplos19/ASPLOS19-LLVM-Tutorial.pdf
- Introduction to smart pointers and move semantics, learncpp.com https://www.learncpp.com/cpp-tutorial/introduction-to-smart-pointers-move-semantics
- 객체의 유일한 소유권 unique_ptr, 모두의 코드 https://modoocode.com/229
- Kaleidoscope: Implementing a Language with LLVM in Objective Caml https://releases.llvm.org/12.0.0/docs/tutorial/index.html
- Clang : a C language family frontend for LLVM https://clang.llvm.org/
- Clang's Documentation https://clang.llvm.org/docs/index.html
- LLVM's Analysis and Transform Passes https://llvm.org/docs/Passes.html
- Difference between clang and gcc https://stackoverflow.com/questions/36542284/difference-between-clang-and-gcc
- 메모리 버그 디텍터 ASAN(Address Sanitizer) 사용법 https://blog.sweetchip.kr/403

이 보고서는 2024년도 한국과학기술정보연구원(KISTI)의
기본사업으로 수행된 연구입니다.
과제번호: (KISTI) K24L2M1C6
과제명: 초고성능컴퓨팅 공동활용을 위한 통합 환경 개발 및 구축

ISBN 978-89-294-1717-8 (PDF)