Metaprogramming

2020-2021 Winter ABC Mentoring Program

Team 5 Mentor 정현준

2021 / 02 / 04

ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY

Metaprogramming이란?

- Meta- 라는 말의 사전적 의미하는 다음과 같다고 합니다.

Metaprogramming

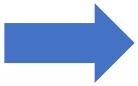
Metadata

Metalanguage

Etc…

Metaprogramming이란?

- 자기 자신, 또는 다른 프로그램을 데이터 취급하여 프로그래밍 하는 것을 의미 (Wikipedia)
- 프로그램을 운영하기 위한 프로그램을 프로그래밍 하는 것

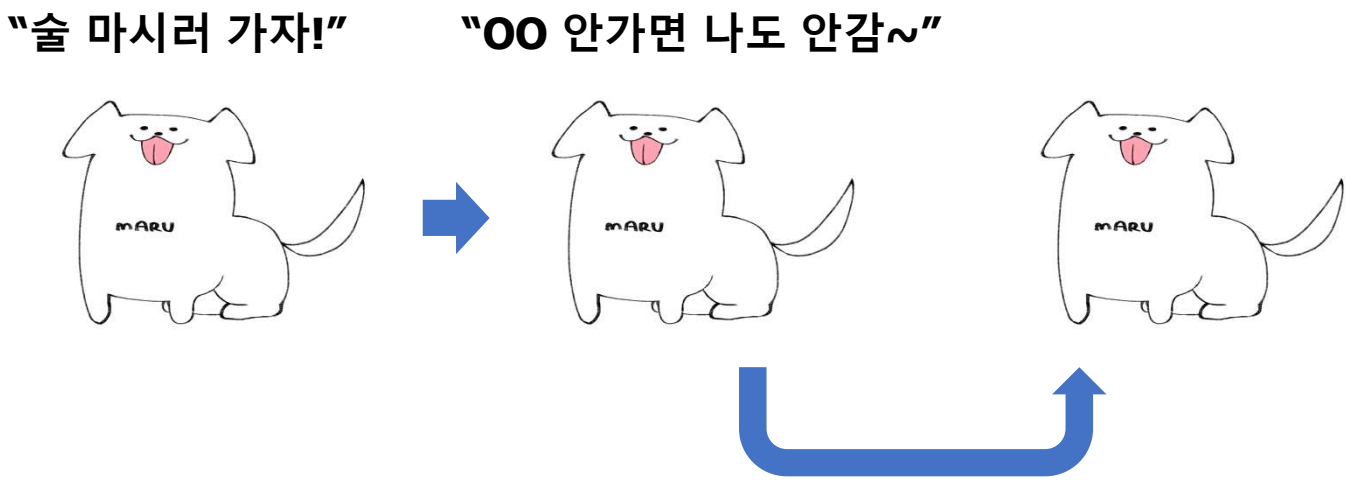


프로그램을 Build 한다

프로그램의 Dependency를 관리한다

Dependency

- "의존성"이라고 불리기도 하며 얼마나 다른 코드에 대해 의존하고 있는가를 나타내는 말 입니다.



Dependency

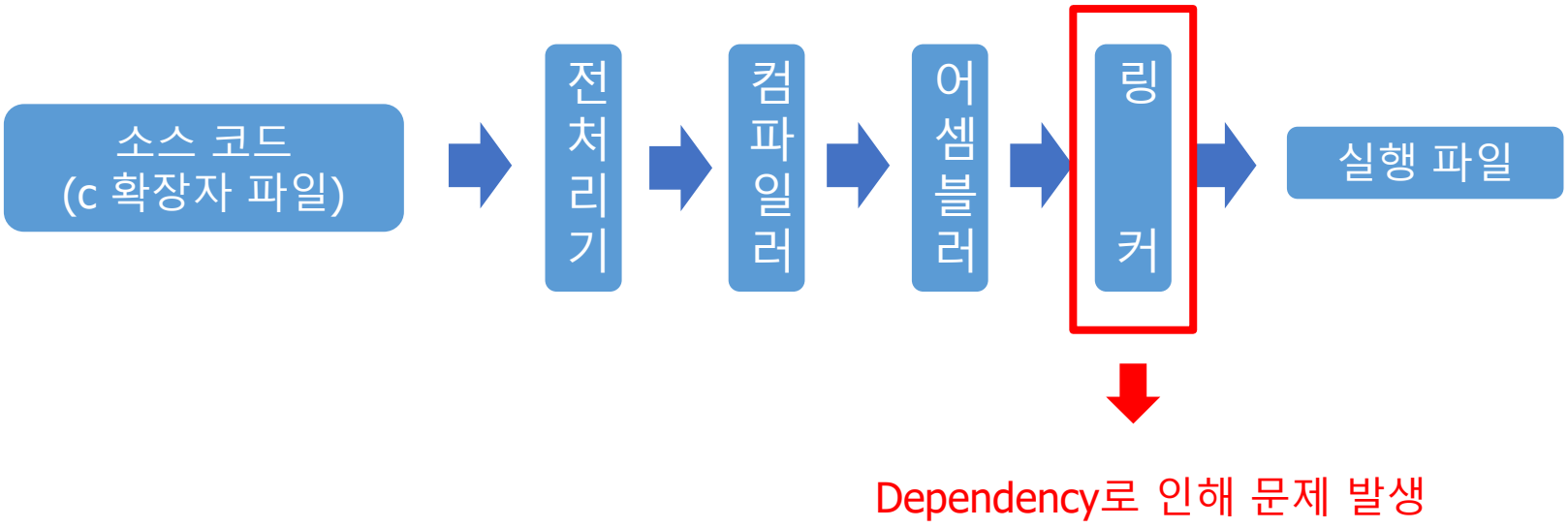
- 프로젝트가 커지면 커질수록 dependency는 더욱 복잡해집니다.
- 따라서 커다란 dependency 관리를 위해 프로그램에 따라 다양한 repository에 접근합니다.
- Ubuntu package => Ubuntu package repository (apt를 이용해 접근)
- Python => PyPi (pip를 이용해 접근)
- Ruby => RubyGems, Java => Maven 등등…
- 이러한 dependency management system은 필요한 dependency 모듈을 버전에 맞게 설치할 수 있는 기능 또한 제공합니다.

Build System

- 프로그래밍에서 "Build"를 한다라는 말은 소스 코드를 실행 가능한 파일로 바꾸는 것을 의미합니다.
- C나 C++에서 쓰이는 "컴파일"은 소스 코드를 바이너리 파일로 바꾸는 과정을 의미하고 이는 빌드의 부분 집합에 해당합니다. (build와의 차이점)
- 소스 코드가 build 되는 과정을 build process라고 합니다.
- 이러한 build process를 도와주는 환경, 또는 tool을 build system이라 합니다.
- 대표적인 build system으로 make가 있습니다.

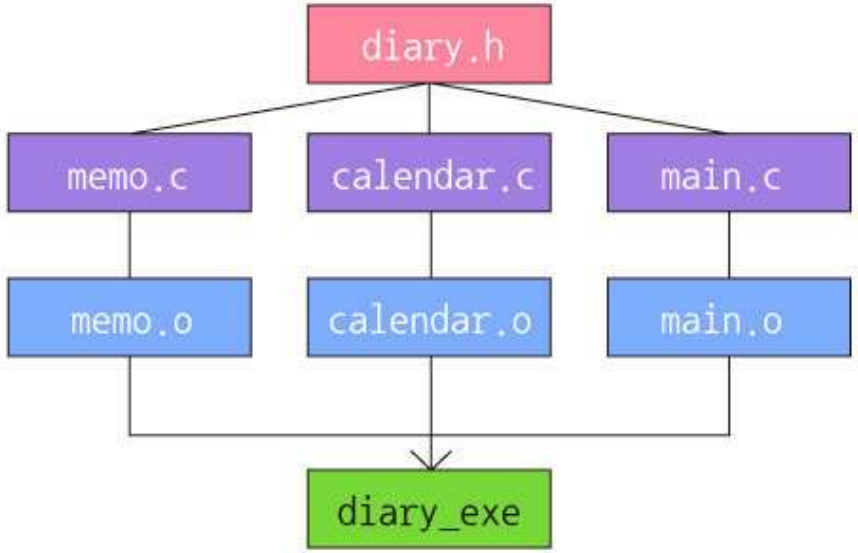
make

- C나 C++에서 주로 사용합니다. (Python에서는 잘 쓰이지 않습니다.)
- UNIX를 기반으로 한 OS에서 거의 설치가 되어 있습니다.
- C 기반 소스 코드가 실행되는 과정



make

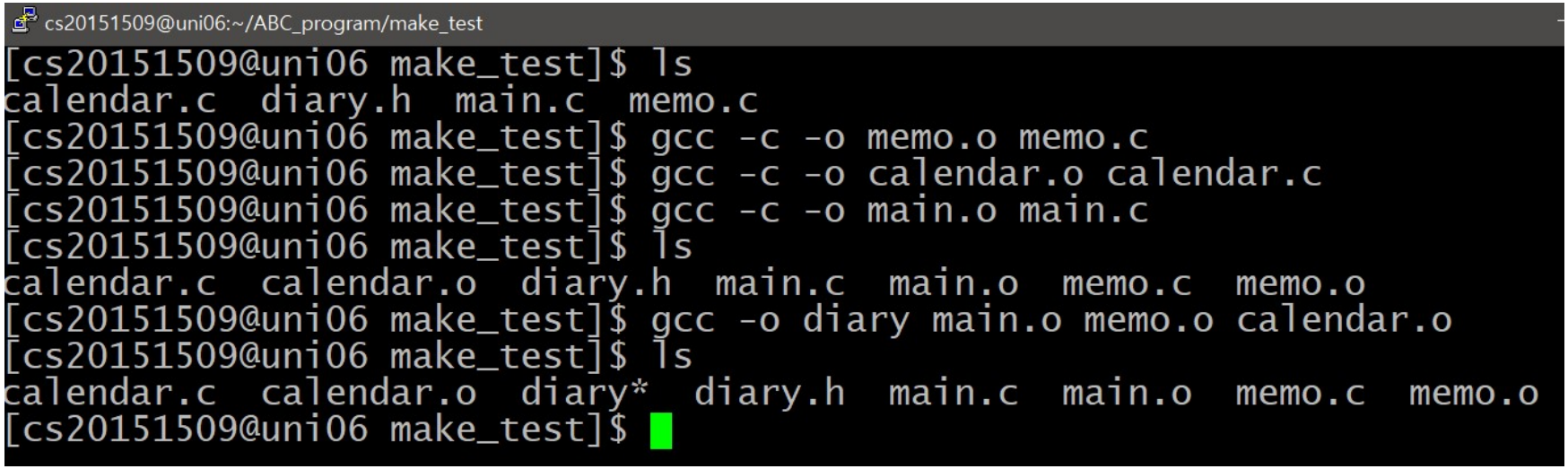
- 각 파일에 대한 반복적인 명령어를 자동화 합니다.
- 프로그램의 dependency 구조를 파악할 수 있고 이를 관리하기 쉬워집니다.
- make를 사용하기 위해서는 Makefile을 작성해야 합니다.



이 파일 dependency 예시를 가지고 make 파일을 작성 해봅시다.

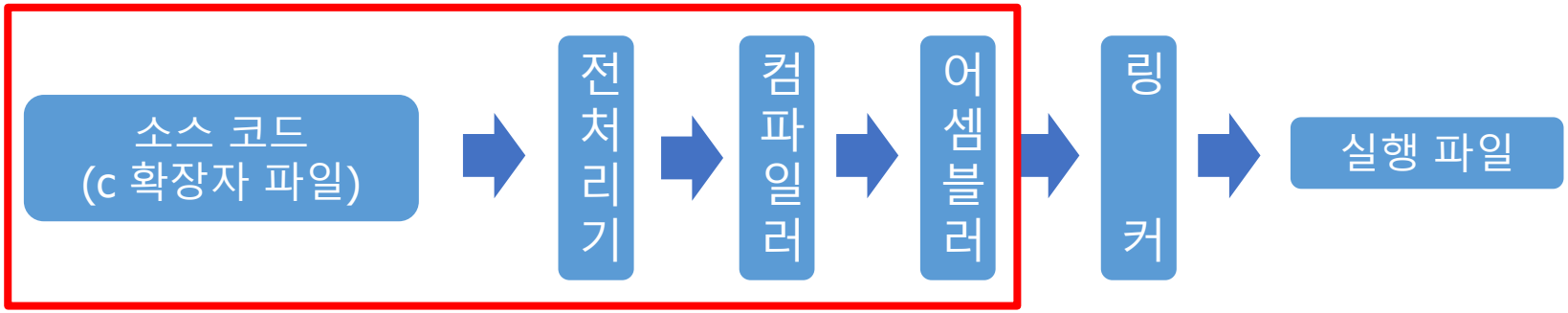
Without make

- Makefile 작성을 하기 전, 만약 make를 사용하지 않고 실행 파일을 만드려면 어떻게 해야 할까요?



Without make

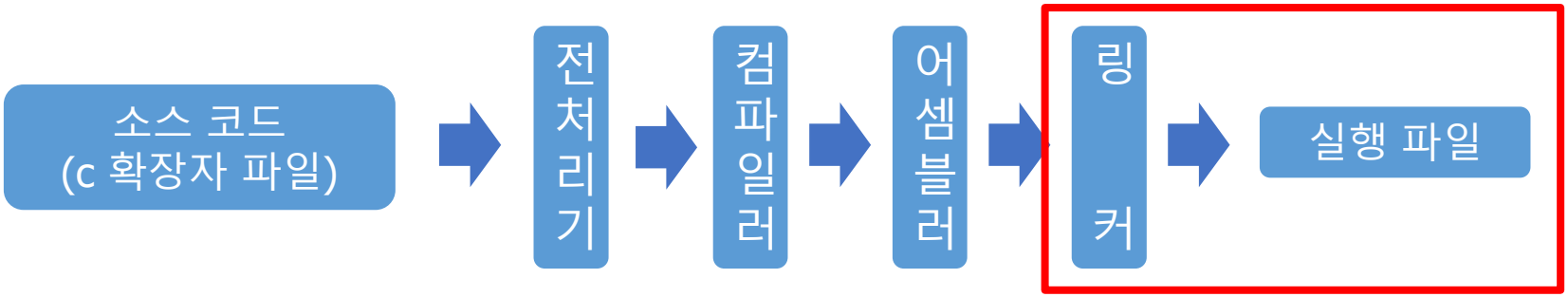
```
[cs20151509@uni06 make_test]$ ls
calendar.c  diary.h  main.c  memo.c
[cs20151509@uni06 make_test]$ gcc -c -o memo.o memo.c
[cs20151509@uni06 make_test]$ gcc -c -o calendar.o calendar.c
[cs20151509@uni06 make_test]$ gcc -c -o main.o main.c
```



- GCC를 실행하면 전처리기부터 어셈블러까지 한꺼번에 실행이 되고, 결과물은 o 파일(object 파일)이 생성됩니다.

Without make

```
[cs20151509@uni06 make_test]$ gcc -o diary main.o memo.o calendar.o
[cs20151509@uni06 make_test]$ ls
calendar.c  calendar.o  diary*  diary.h  main.c  main.o  memo.c  memo.o
[cs20151509@uni06 make_test]$
```



- 생성된 object 파일은 기계가 이해할 수 있는 기계어로 작성된 파일이지만 dependency 연결이 되지 않은 상태입니다.
- 따라서 다시 gcc를 통해 링커 과정을 거쳐야 합니다.

Makefile

- 앞서 예제의 경우, 파일이 몇 개 되지 않기 때문에 명령어를 사용하는 과정이 어렵지 않았습니다.
- 하지만 dependency 파일이 많아지면 이러한 과정이 매우 힘들 것입니다.
- Makefile의 작성 규칙은 다음과 같습니다.

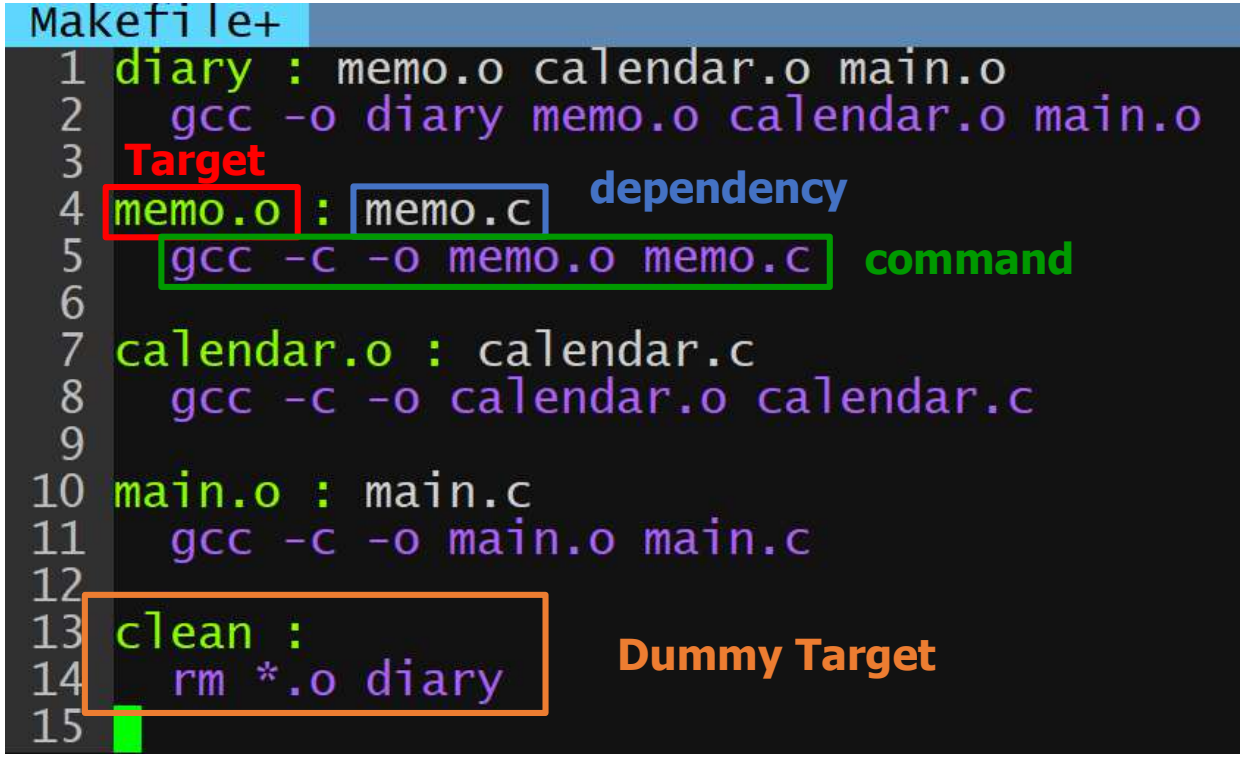
```
Target1 : dependency1 dependency2 …
        command 1
        command 2
        …

Target2 : dependency1 dependency2 …
        command 1
        command 2
        …
```

Makefile

- Target은 만들고자 하는 어떠한 목표 파일을 의미 합니다.
- Dependency는 target을 만드는데 필요한 dependency 파일들을 의미합니다.
- Command는 dependency를 이용해 target을 만드는데 사용하는 명령어 입니다.
- 주의할 점으로 command를 사용하기 위해서는 Python처럼 tab을 이용한 들여쓰기가 필요합니다.
- 또한 일부 target에 대해서는 dependency가 존재하지 않을 수도 있습니다.
- 링커는 가장 맨 첫 줄에 쓰여야 합니다.

Makefile 예시



- Dummy target은 파일을 생성하지 않지만 make에 특별한 기능을 부여합니다.
- make clean을 입력하게 되면 dummy target의 명령어가 실행됩니다.

Macro

Macro

```
1  CC = gcc
2  CFLAGS = -W -Wall
3  TARGET = diary
4
5  $(TARGET) : memo.o calendar.o main.o
6    $(CC) $(CFLAGS) -o $(TARGET) memo.o calendar.o main.o
7
8  memo.o : memo.c
9    $(CC) $(CFLAGS) -c -o memo.o memo.c
10
11 calendar.o : calendar.c
12   $(CC) $(CFLAGS) -c -o calendar.o calendar.c
13
14 main.o : main.c
15   $(CC) $(CFLAGS) -c -o main.o main.c
16
17 clean :
18   rm *.o $(TARGET)
```

- Makefile 안에서 변수와 같은 역할을 합니다.

Macro

- Macro는 반드시 치환 될 위치보다 먼저 정의되어야 합니다.
- Macro를 사용할 때는 $(Macro 이름), 즉 소괄호화 $ 기호가 필수 입니다.
- 아래 변수명은 Make 자체 내장 변수 입니다.
- 자체 내장 변수는 아래와 같은 지정된 용도로 사용되어야 합니다.

CC	사용할 컴파일러
CFLAGS	컴파일러에 들어갈 옵션
OBJS	중간 object 파일 목록
TARGET	빌드 대상 이름. 즉, 실행 파일 이름
LDFLAGS	링커에 사용될 옵션
LDLIBS	링크 라이브러리 목록

Automatic Variable

```
1  CC = gcc
2  CFLAGS = -W -Wall
3  TARGET = diary
4  OBJS = memo.o main.o calendar.o
5
6  all : $(TARGET)
7
8  $(TARGET) : $(OBJS)
9    $(CC) $(CFLAGS) -o $@ $^
10
11 clean :
12   rm *.o $(TARGET)
```

- $@은 현재 Target의 이름을 의미합니다.
- $^은 현재 Target의 dependency 항목 리스트를 나타냅니다.

Makefile

```
1  CC = gcc
2  CFLAGS = -W -Wall
3  TARGET = diary
4  OBJS = memo.o main.o calendar.o
5
6  all : $(TARGET)
7
8  $(TARGET) : $(OBJS)
9    $(CC) $(CFLAGS) -o $@ $^
10
11 clean :
12   rm *.o $(TARGET)
```

- all은 clean처럼 dummy target으로, target을 만들기 위한 dependency를 확인합니다. 주로 targe이 여러 개일 때 사용합니다.
- o 파일을 만드는 과정이 명시되어 있지 않은 경우, 같은 이름의 c 파일을 찾아 자동으로 생성하는 과정을 진행합니다.

Program Test

```
if X < Y :
        print ("X는 Y보다 작다.")
elif X > Y :
        print ("X는 Y보다 크다.")
else :
        print ("X와 Y는 같다.)
```

If문을 테스트 하려면 X와 Y에
어떤 값들이 들어가야 할까요?

- 완성된 프로젝트는 배포가 되기 전에 반드시 검증이 되어야 합니다.
- 검증을 하는 과정을 Test라 하고, Test에 사용되는 Test 값 집합을 "Test Suite" 라고 합니다.

Program Test

- 적은 수의 test case를 가지고 모든 경우의 수를 검증할 수 있는 적절한 test suite를 만드는 것이 중요합니다.
- Unit test => 하나의 함수나 명령문을 검증
- Integration test => unit test가 모여서 더 큰 부분을 검증
- 그 외 다양한 test 테크닉이 존재합니다. (유전알고리즘, 강화학습 등등…)
- 자세한 내용은 CSE36401 소프트웨어 공학 과목에서 들으실 수 있습니다.

Assignment

- 오늘 배운 내용을 토대로 직접 Makefile을 만들어 봅시다.
- https://github.com/with1015/2020_ABC_Mentoring/tree/master/makefile_test
- 만든 Makefile을 Github에 올려 봅시다.

- 다음 강의 : Security and Cryptography
- https://missing.csail.mit.edu/2020/security/

Thank you


UNIST
ULSAN NATIONAL INSTITUTE OF
SCIENCE AND TECHNOLOGY
2009