10 Makefile

Duże projekty programistyczne to kolekcja setek plików które należy przetworzyć lub skompilować w pewnej określonej kolejności. Robienie tego ręcznie, czy nawet w skrypcie, wymagałoby sporego nakładu pracy. W praktyce robi się to w specjalnym języku stworzonym do tego celu. Dwa najbardziej popularne rozwiązania to:

* plik `Makefile` pisany w języku `make`
* plik `CMakeLists.txt` pisany w języku `cmake`

Poniżej skupimy się na prostszym (i starszym) rozwiązaniu jakim jest `Makefile`.

Reguły

Plik `Makefile` składa się z *reguł*. Każda reguła to fragment kodu w bashu, który przetwarza jakieś pliki wejściowe i produkuje pliki wyjściowe. Regułę zapisujemy:

```
<plik_wyjsciowy> : <pliki_wejsciowe>
        <komendy>
```

W danej regule może być umieszczony dowolnie długi skrypt, ale każda linia musi być rozpoczęta znakiem tabulacji (wcięcia ze spacji nie zadziałają!). Ponadto, każda linia która jest wykonana podczas kompilacji jest wyświetlana na ekran. W ten sposób programista może kontrolować co się dzieje. Rozpoczęcie linii poprzez `@` powoduje jej ukrycie.

* plik wyjściowy nazywa się *celem* (ang. *target*), lub po prostu targetem.
* każdy plik wejściowy to *niezbędnik* (ang. *prerequisite*). Mówimy, że target jest *zależny od* tych plików niezbędnych. Pliki wejściowe można definiować używając globów.

Załóżmy, ze kompilujemy pliki `main.c` i `nwd.c` do programu `prog`. Ponadto, mamy plik nagłówka `nwd.h`, który jest wykorzystywany w obu plikach `.c`. Przykładowy `Makefile`:

```
prog : nwd.o main.o
        gcc nwd.o main.o -o prog

nwd.o : nwd.c nwd.h
        gcc nwd.c -c -o nwd.o

main.o : main.c nwd.h
        gcc main.c -c -o main.o
```

Gdy w wierszu poleceń zostanie uruchomiona komenda `make`, program wyszukuje plik o nazwie `makefile` lub `Makefile` w aktualnym katalogu. Pierwsza reguła w `Makefile`-u jest regułą domyślną i to ona zostaje wykonana przy uruchomieniu pliku. Uruchomienie konkretnej reguły: `make <target_name>`. Użycie innego pliku: `make -f <plik_make>`. Można wywołać wiele reguł `make target1 target2 ...`

Gdy reguła zostaje uruchomiona:

* Sprawadzane są wszystkie pliki wejściowe. Jeśli dla nich istnieją reguły, zostają one rekurenyjnie uruchomione.

* Porównywane są czasy ostatnich zmian w plikach. Jeśli target jest nowszy niż wszystkie jego niezbędniki, nic się nie dzieje.
* Jeśli target jest starszy od któregoś z plików wejścowych, lub po prostu nie istnieje, wykonywane są wszystkie komendy danej reguły.

Dla dużych projektów, mechanizm wykluczający reguły gdzie target jest aktualny względem jego niezbędników pozwala na znaczne przyspieszenie procesu rekompilacji, gdy zmianie uległo niewiele plików. Tam gdzie nie ma zmian w plikach źródłowych, starsze wersje plików `.o` mogą zostać natychmiast wykorzystane.

Ważnym jest jednak, by w liście plików wejściowych uwzględnić wszystkie pliki od których target zależy, nawet jeśli podane nazwy nie pojawiają się bezpośrednio w komendach. Dlatego na przykład uwzględniliśmy plik nagłówkowy `nwd.h`.

Specjalne cele wymienione pod targetem `.PHONY` są wykluczone z porównywania czasów – komendy w niej zawarte zawsze się wykonają. Najczęściej nazwa takiego celu nie jest powiązana z żadnym fizycznym plikiem, a jest tylko nazwą pewnej procedury. Typowym celem tego typu jest `all`, który zwykle oznacza wykonanie pełnej kompilacji/instalacji.

```
all : prog1 prog2
.PHONY: all

prog1 : prog1.o lib.o
        gcc prog1.o lib.o -o prog1

prog2 : prog2.o lib.o
        gcc prog2.o lib.o -o prog2
...
```

W powyższym przykładzie target `all` oznacza wywołanie `prog1` i `prog2` czyli skompilowanie dwóch programów zawartych w tym samym projekcie. Reguła `all` jest wymieniona jako pierwsza, więc to ona wywoła się, gdy uruchomimy `make` bez argumentów. Możemy też wywołać osobno `make prog1` albo `make prog2` by skompilować wskazany program.

Zmienne

Zmienne w `Makefile` wyglądają podobnie do zmiennych w bashu. Trzeba jednak pamiętać, że są to dwa zupełnie różne byty.
Zmienne `Makefile`-a mogą pojawić się gdziekolwiek: są definiowane poza regułami i mogą pojawić się zarówno jako pliki wejściowe/wyjściowe jak i w obrębie komend reguły. Definiuje się je zwykle na początku skryptu, przed regułami (bez wcięć).

```
nazwa := wartosc     # przypisanie statyczne
nazwa2 = wartosc     # przypisanie rekursywne
nazwa3 ?= wartosc    # przypisanie warunkowe
nazwa3 += wartosc    # dopisanie
```

Użycie zmiennej: `$(nazwa)` – nawiasy można pominąć jedynie jeśli nazwa jest jednoliterowa.

Zmienne statyczne i rekursywne różnią się tym kiedy rekurencyjne odwołania do zmiennych zostają zastąpione ich wartością. Wartość **w zmiennej statycznej** jest rozwinięta podczas definicji i może odwoływać się tylko do innych zmiennych zdefiniowanych wcześniej. **Zmienne rozwijane rekursywnie** zapisują całe wyrażenie w postaci źródłowej i wartość jest rozwinięta do postaci końcowej dopiero wtedy gdy zmienna jest użyta w końcowym wyrażeniu. To pozwala takiej zmiennej na zawieranie odwołań które w momencie definicji jeszcze nie istnieją.

```
nazwa=main.c
plik=src/$(nazwa)
nazwa=nwd.c

all :
        echo $(plik)
        #wypisze src/nwd.c

.PHONY: all
```

```
nazwa=main.c
plik:=src/$(nazwa)
nazwa=nwd.c

all :
        echo $(plik)
        #wypisze src/main.c

.PHONY: all
```

Przypisanie warunkowe definiuje nową wartość jedynie pod warunkiem, że podana nazwa zmiennej nie istnieje.
Operacja dopisania wydłuża zawartość danej zmiennej o podaną nową treść. Typ zmiennej (statyczna/rekursywna) nie ulega zmianie. W przypadku zmiennej statycznej jest to równoważne z napisaniem:

```
nazwa := A
nazwa := $(nazwa) B
#nazwa == A B
```

Natomiast gdybyśmy tak zrobili w przypadku zmiennej rekursywnej (= zamiast :=), `Makefile` zapętlił by się w momencie użycia tej zmiennej:

```
#$(nazwa)  ->  $(nazwa) B  ->  $(nazwa) B B  ->  $(nazwa) B B B  -> ...
```

Dlatego użycie += jest bezpieczniejsze i krótsze.

Ważne! Możemy nadpisać/zdefiniować wartość zmiennej w momencie uruchomienia:
`make CC=g++ CFLAGS="-O3 -Wall" <target>`

Zmienne automatyczne

Wewnątrz reguł, `Makefile` definiuje serię zmiennych automatycznych. Przy ich pomocy reguły działające co do zasady w ten sam sposób, ale operujące na różnych plikach mogą wyglądać identycznie.

- `$@` Nazwa aktualnego targetu
- `$*` Fragment dopasowany do `%` (o czym za chwilę)
- `$<` Nazwa pierwszego niezbędnika
- `$?` Nazwy wszystkich niezbędników nowszych niż cel, oddzielone spacją
- `$^` Nazwy wszystkich niezbędników. Każda nazwa występuje maksymalnie jeden raz
- `$+` Nazwy wszystkich niezbędników, w kolejności i liczbie dokładnie jak w `Makefile`

Każda z powyższych zmiennych może odnosić się do pliku znajdującego się w innym katalogu. Jeśli tak jest, to powyższa nazwa odpowiada ścieżce dostępu do danego pliku. Można jednak łatwo uzyskać sam katalog, albo samą nazwę pliku bez katalogu, pisząc odpowiednio `$(XD)` albo `$(XF)`, gdzie `X` to odpowiedni znak z powyższej tabeli. Np. `$(@D)` to katalog aktualnego targetu.

Wyposażeni w takie narzędzia możemy tworzyć reguły bardziej uniwersalne:

```
h_files:=nwd.h
obj_files:=nwd.o main.o

nwd : $(obj_files)
        gcc $+ -o nwd

%.o : %.c $(h_files)
        gcc $< -c -o $@
```

Target używający w swojej nazwie znaku % to bardzo prosty pattern, gdzie % dopasowuje się do dowolnego niepustego ciągu znaków. Taki sam znak % w niezbędniku to odwołanie do ciągu znaków który został uprzednio dopasowany w targecie. Tak więc, w kontekście powyższego przykładu, `%.o : %.c $(h_files)` dopasowuje się dwukrotnie, tworząc dwie reguły:

* `main.o : main.c nwd.h`
* `nwd.o : nwd.c nwd.h`

W każdej z tych reguł, wartości pod zmiennymi `$+`, `$<`, `$@` są inne – odpowiadające aktualnej wersji reguły.

Wszystkie zmienne tu omawiane są obsługiwane przez język `make`. Gdy taka zmienna pojawia się w treści komendy, zostaje ona zastąpiona jej właściwą wartością, i tak przetworzone wyrażenie jest przekazane do basha. Jeśli wewnątrz komendy potrzebujemy użyć zmiennych obsługiwanych przez basha, symbol `$` trzeba napisać dwukrotnie.

Dla przykładu, poniżej reguła `test.out` uruchamia program `prog` 5 razy i wylicza średni rzeczywisty czas jaki został poświęcony na działanie programu, zapisując wynik do `test.out`. Zmienna `$(TIME)` jest obsługiwana przez `Makefile` i wymaga pojedyńczego znaku `$`. Natomiast zmienne `$time` oraz `$value`, jak i wyrażenia `$(...)` są operacjami wewnątrz basha, nie kontrolowanymi przez `Makefile` – i one potrzebują `$$`.

```
TIME := /usr/bin/time

test.out : prog
	time=0
	for i in {1..5}; do
	  value="$$($(TIME) -f %e prog 2>&1 )"
	  time="$$(echo "$$time+$$value" | bc -l)"
	done
	echo "$$time/5" | bc -l > test.out
```