make ו- makefile

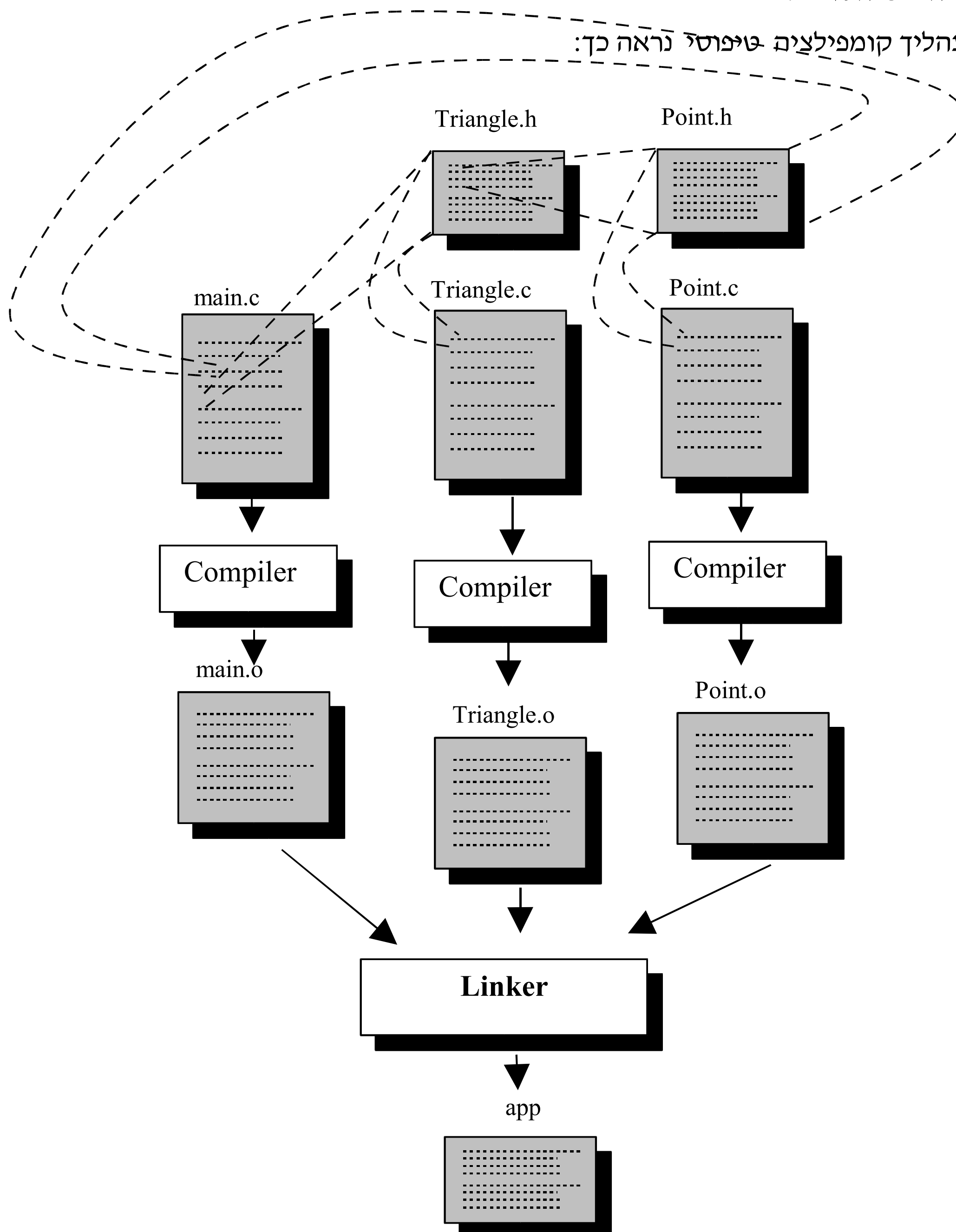
make

make הוא כלי (כלומר תוכנה המותקנת תחת מערכת ההפעלה) של UNIX המאפשר תזמון של פעולות שונות על קבצים. בחירת הפעולות וסדרן תלויים בתאריכי הקבצים. אנחנו נשתמש ב – make ליצירת תהליך קומפילציה אוטומטי של פרוייקט מרובה קבצים.

מוטיבציה

תהליך קומפילציה של פרוייקט מרובה קבצים כרוך, כידוע, בכמה הפעלות של קומפיילר (שכל אחת מהן מתחילה בפעולת preprocessor) וכן הפעלה של linker המקשר בין תוצרי הקומפילציה של קבצי ה – C השונים.

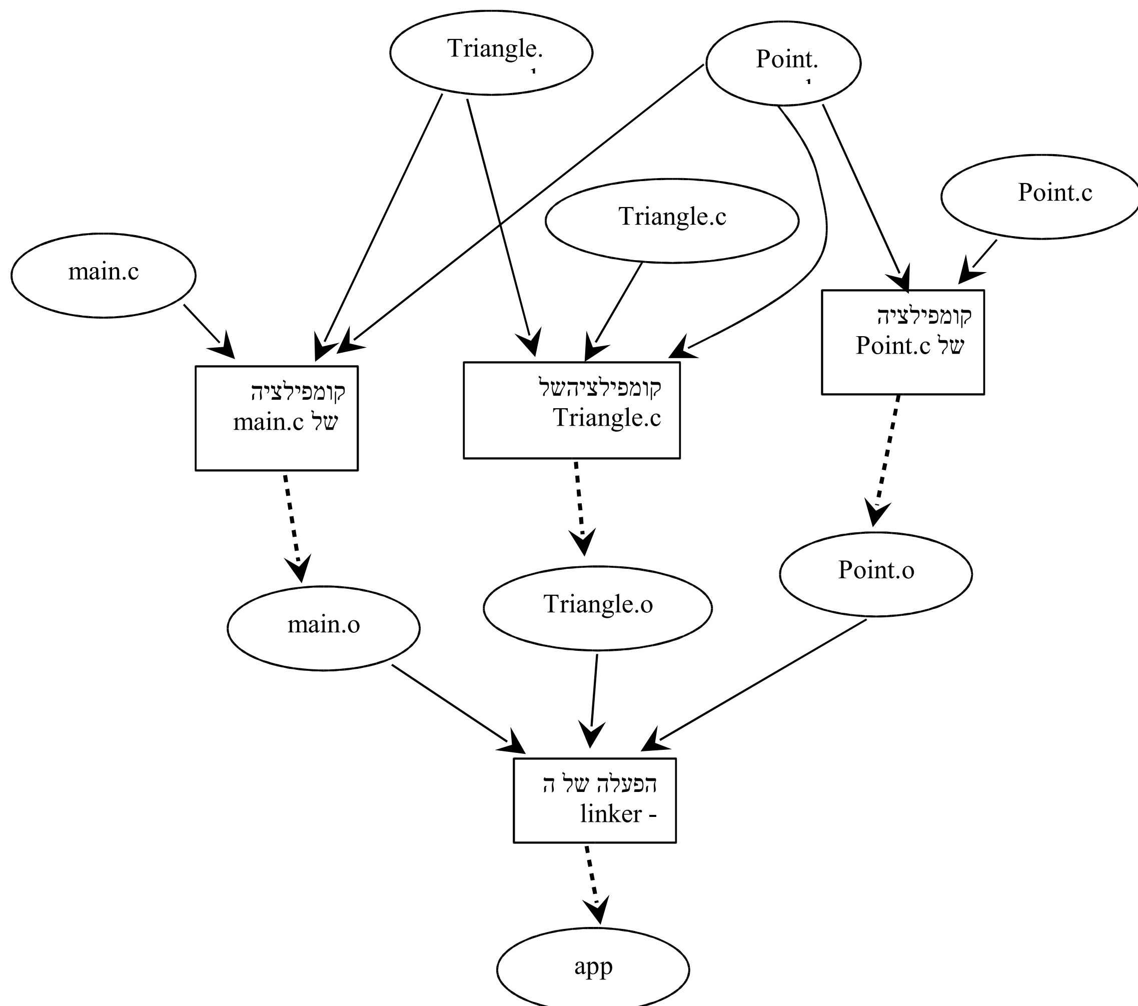
כזכור, תהליך קומפילצים טיפוסי נראה כך:



מהסכמה אפשר לראות שקומפלציה של כל הפרוייקט כוללת הפעלה של הקומפיילר עבור כל אחד מקבצי ה – C וכאשר מסתיימות כולן – הפעלה של ה – linker המקשר את כל תוצרי הקומפילציה.

כיוון שקומפילציה היא תהליך הדורש זמן, נשאלת השאלה האם אנו זקוקים לכל התהליך שפורט, כל פעם שנבצע שינוי באחד הקבצים. אם נשנה את Triangle.c האם נאלץ לקמפל מחדש את main.c ואת Point.c ? או אם נשנה את Point.h איזה קבצים נצטרך לקמפל מחדש ?

ננסה לתאר את תלות הפעולות השונות בשינויי קבצים:

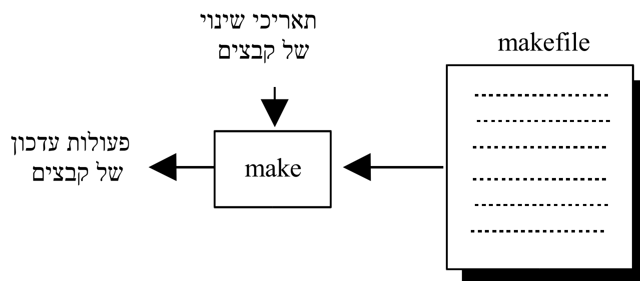


חץ רגיל לפעולה, משמעו ששנוי בקובץ גורר הפעלה של הפעולה. חץ מקוקו מפעולה, משמעו שהפעלת הפעולה משנה את הקובץ (או יוצרת אותו).

כפי שרואים, נוצר גרף מכוון חסר מעגלים (DAG) של תלויות. הגרף חייב להיות חסר מעגלים כי אחרת שינוי של קובץ היה גורם לשינוי באותו הקובץ, דבר שיגרום לקומפילציה אין סופית.
כל שינוי של קובץ יגרור אחריו הפעלת כל הפעולות הנמצאות בתת הגרף שמתחתיו. שינוי של מספר קבצים יגרור את הפעלת איחוד ההפעלות של כל אחד בנפרד.

makefile

makefile הוא הקובץ אותו מקבלת התוכנה make כקלט. קובץ זה מתאר את גרף התלויות שתיארנו בדרך גראפית. לפי האינפורמציה שיש ב – makefile ולפי התאריכים בהם שונו הקבצים, יודע make אילו פעולות יש לבצע על מנת לעדכן את הקבצים הדרושים.



ל – makefile יש פורמט קבוע, הבנוי ממקבצים של הוראות:

makefile:

```
file: file1 file2 … fileN
        shell-command1
        shell-command2
        …
```

```
file: file1 file2 … fileN
        shell-command1
        shell-command2
…
```

(בדוגמה הזו רואים שני מקבצים)

כל מקבץ טקסטואלי כזה מייצג פעולה (או הוראה למערכת ההפעלה) ואת התלויות שלה בקבצים אחרים. מקבץ-פקודה כזה בנוי משורת תלויות:

file: file1 file2 … fileN

משמעותה של השורה היא שהפקדות שבמקבץ יפעלו אם"ם תאריך השינוי של אחד מהקבצים file1,..,fileN הנו **מאוחר יותר** מתאריך השינוי של file. שאר השורות במקבץ הן פקודות לביצוע בתנאי שהתנאי מתקיים. שורות פקודה אלו אמורות לעדכן את file בהתאם לשינויים שנערכו בקבצים file1,..,fileN. יש לשים לב שכל שורת פקודה חייבת להתחיל ברווח – tab.

ניראה את ה – makefile המתאים לדוגמה בה עסקנו:

makefile:

```
app: main.o Triangle.o Point.o
      gcc main.o Triangle.o Point.o -o app


main.o: main.c Triangle.h Point.h
      gcc -c main.c

Triangle.o: Triangle.c Triangle.h Point.h
      gcc -c Triangle.c

Point.o: Point.c Point.h
      gcc -c Point.c
```

אפשר לראות את הקוד הזה כקידוד של הגרף אותו הראנו קודם. את קובץ ה – makefile אפשר למקם ב – directory שבו נמצאים קבצי הפרוייקט ואז הפקודה make מבצעת את הקומפילציה המינימלית הדרושה. ניתן לקרוא לקובץ makefile או Makefile. אפשר גם לקרוא לו בשם אחר אך אז יש לציין שם זה בפקודת הפעלת ה - make.

למעשה התוכנית make מבצעת אלגוריתם פשוט של מיון טופולוגי. על הקוד בקובץ ה – makefile אין לחשוב כעל תוכנית או אלגוריתם אלא כעל **תאור** של אוסף תלויות הדדיות המשמש את make.

ניתן להוסיף כל פקודת shell נוספת לתהליך, לדוגמה:

```
app: main.o Triangle.o Point.o
      ls
      echo this text will be displayed
      gcc main.o Triangle.o Point.o -o application
```

כמו כן ניתן להשתמש במשתנים המחליפים מחרוזות אותיות, לדוגמה:

```
CC = gcc
OBJECTS = main.o Triangle.o Point.o

app: $(OBJECTS)
	$(CC) $(OBJECTS) -o app
```

ההצהרה ואתחול של המשתנה היא ע"י שימוש ב – 'Var = string'. כדי להשתמש בערך המשתנה משתמשים ב - $(Var).

הפעלת make ללא פרמטרים מבטיחה עדכון של הקובץ הראשון ב makefile וכל הקבצים בהם הוא תלוי. ניתן להוסיף כפרמטר את הקובץ אותו רוצים לעדכן. לדוגמה: make main.o בדוגמה בה עסקנו, יבטיח קומפילציה מחודשת של main.c אבל לא יבצע את פעולת ה – linking.

ניתן לבנות פרוייקט בו יכולים להיווצר מספר קבצי ריצה. ניתן למשל להוסיף לדוגמה שלנו עד קובץ עם main:

file otherMain.c:

```
#include "Point.h"
int main() {
	// testing the Point class ..
}
```

ואז להוסיף ל makefile את השורות הבאות:

```
otherMain.o: otherMain.cc Point.h
	gcc -Wall –c otherMain.cc
pointTest: otherMain.o Point.o
	gcc -Wall otherMain.o Point.o –o pointTest
```

כעת ניתן לכתוב: make pointTest שיצור את תוכנית הבדיקה של point וכן ניתן לכתוב make app שיצור את התוכנית הקודמת. למרות שמדובר בשני קבצי הרצה שונים, אם לצורך קומפילציה של האחד, Point.c קומפל, אין צורך לקמפל אותו מחדש עבור השני.

שימו לב כי אין כאן סתירה לכלל שכל תוכנית אמורה לכלול main יחיד מכיוון שמדובר פה בשתי תוכניות שונות באותו פרוייקט.

המידע שהוצג עד כה מספיק בכדי לכתוב makefile לפרוייקטים קטנים. בשלב זה אתם יכולים לכתוב פעם אחת מספר שורות המתארות תלויות ופעולות קומפילציה וליהנות מקומפילציה אוטומטית. בפרוייקטים גדולים יותר, כתיבת ה - makefile באופן שהוצג כאן עלולה להיות עבודה מייגעת ואף לגרור טעויות. לשם כך קיימים כלים המבצעים חלק ניכר מהמטלות של כתיבת ה - makefile באופן אוטומטי. אנו נציג כלים אלו מיד אך נציין שכדאי לכתוב מספר makefile-ים פשוטים ולהבין היטב כיצד הם עובדים לפני שמתחילים להשתמש בכלים

האוטומטיים. הכלים האוטומטיים עלולים להיות מבלבלים ושגיאה ב - makefile עלולה לגרום לאי-קימפול של קובץ חיוני וליצירת אפליקציה שאיננה תואמת את הקוד שלכם.

makefile עם חוקים כלליים וחוקים הנוצרים אוטומטית

נניח שהפרוייקט שלנו כולל את הקבצים הבאים:

```
file1.c:
	#include “file1.h”
	int main() {…}
file1.h:
	#include “defs.h”
	…
defs.h:
	#include “global.h”
	…
global.h:
	…
file2.c:
	  #include “defs.h”
	int main() {…}
utils.c:
	#include “file1.h”
	…
```

לפי ההסבר שהוצג עד כה, היינו כותבים לפרוייקט זה את ה - makefile הבא:

```
all: prog1 prog2

file1.o: file1.c file1.h defs.h global.h
	gcc -Wall file1.c
file2.o: file2.c defs.h global.h
	gcc -Wall file2.c
utils.o: utils.c file1.h defs.h global.h
	gcc -Wall utils.c
prog1: file1.o utils.o
	gcc file1.o utils.o –o prog1
```

```
prog2: file2.o utils.o
	gcc  file2.o utils.o –o prog2
```

כל שלושת החוקים המתייחסים לקומפילציה דרשו עבודה טכנית החוזרת על עצמה ומזמינה שגיאות. מאד קל לשכוח להכליל את global.h לדוגמה ברשימת התלויות. במיקרה של שיכחה כזו, כאשר global.h ישונה, לא תתבצע קומפילציה מחודשת כנדרש וייווצר bug קשה לאיתור. אנחנו נרצה כלי אוטומטי שימצא את כל הקבצים שבהם הקומפילציה תלויה.

ראשית, נציין כי חוקי התלות בין הקבצים יכולים להופיע בפני עצמם. את חוק הקומפילציה הראשון לדוגמה יכולנו גם לכתוב ע"י אוסף החוקים הבאים:

```
file1.o: file1.c
	gcc -Wall file1.c
file1.o: file1.h defs.h global.h
```

או:

```
file1.o:
	gcc -Wall file1.c
file1.o: file1.h defs.h global.h
file1.o: file1.c defs.h
```

(העובדה שתלות מצוינת יותר מפעם אחת אינה מהווה בעיה)

בשביל למצוא את כל הקבצים בהם תלויה הקומפילציה של file1.c . נוכל להשתמש ב – preprocessor של C. הפקודה הבאה מה – command-line :

```
$ gcc -MM file1.c
```

תדפיס לפלט הסטנדרטי את חוק התלות של file1.c:

```
file1.o: file1.c file1.h defs.h global.h
```

כאשר חושבים על כך, ל – preprocessor לא צריך להיות קשה במיוחד למצוא את חוק התלות מכיוון שגם כאשר הוא פועל את פעולתו הרגילה, הוא צריך למצוא את כל הקבצים הנכללים באופן עקיף ב – file1.c.

מאחר ויש לנו תוכנה המייצרת את חוקי התלות לקומפילציה באופן אוטומטי, ניתן לכתוב את ה – makefile באופן הבא:

```
all: prog1 prog2

file1.o
	gcc -Wall file1.c
file2.o:
	gcc -Wall file2.c
utils.o:
	gcc -Wall utils.c
prog1: file1.o utils.o
	gcc file1.o utils.o –o prog1
prog2: file2.o utils.o
	gcc file2.o utils.o –o prog2
```

ואז להריץ את הפקודה הבאה מה – command-line:

```
gcc -MM *.c >> makefile
```

הפקודה הזאת תייצר את חוקי התלות של כל קבצי ה - c. ותשרשר אותם בסוף קובץ ה – makefile. לאחר הרצת הפקודה הנ"ל, יראה קובץ ה – makefile שלנו כך:

```
all: prog1 prog2

file1.o
	gcc -Wall file1.c
file2.o:
	gcc -Wall file2.c
utils.o:
	gcc -Wall utils.c
prog1: file1.o utils.o
	gcc -Wall file1.o utils.o –o prog1
prog2: file2.o utils.o
	gcc -Wall file2.o utils.o –o prog2
file1.o: file1.c file1.h defs.h global.h
file2.o: file2.c defs.h global.h
```

```
utils.o: utils.c file1.h defs.h global.h
```

כאשר אנו משתמשים בכלי האוטומטי, אנו חוסכים עבודה וסיכוי לטעות.

ניתן לבדוק את נכונות קובץ ה – makefile בעזרת הפקודה touch. כאשר כותבים לדוגמה 'touch file1.h' מה- command-line, ישתנה תאריך השינוי של file1.h לזמן הנוכחי. מבחינת תהליך הקומפילציה , הקובץ file1.h שונה ויש לקמפל מחדש את כל הקבצים התלויים בו. הרצה נוספת של make תאפשר לנו לבדוק אם אכן בוצעו הפעולות הדרושות.

כעת, כשמסתכלים על ה – makefile נראים כל החוקים המתארים את פקודת הקומפילציה עצמה, מאד דומים זה לזה. טבעי לשאול מדוע עלינו לכתוב כל כך הרבה חוקים המתארים בעצם חוק כללי: בשביל לצור קובץ .o יש לקמפל קובץ .c מתאים לפי פקודת קומפילציה מסוימת.
ואכן, make מאפשר לכתוב חוקים כלליים כאלה. ניתן לכתוב את החוק הדרוש באופן הבא:

```
.c.o:
	gcc -Wall –c $<
```

הסימון ".c.o" מציין שמדובר בחוק כללי להפקת קובץ .o מקובץ .c הסימון '$<' הוא macro (כלומר איזו מחרוזת של תווים שתחליף את הסימן) של אותו שם קובץ שגרם להפעלת החוק. במקרה זה, '$<' יהיה שם אותו קובץ .c שאותו יש לקמפל ולייצר את קובץ ה - .o המתאים.

כעת ה- makefile שלנו, הצטמק ל:

```
all: prog1 prog2
.c.o:
	gcc -Wall –c $<
prog1: file1.o utils.o
	gcc -Wall file1.o utils.o –o prog1
prog2: file2.o utils.o
	gcc -Wall file2.o utils.o –o prog2
file1.o: file1.c file1.h defs.h global.h
file2.o: file2.c defs.h global.h
utils.o: utils.c file1.h defs.h global.h
```

כאשר את שלושת הפקודות האחרונות לא היינו צריכים לכתוב בעצמנו.

נהוג להוסיף target נוסף ל – makefile שיפעיל את הכלי המוסיף את כללי התלות. הגרסה הבאה כוללת target כזה ובנוסף, משתמשת ב macro-ים המוגדרים על ידינו:

```
CC = gcc                    # the compiler
CFLAGS = -Wall              # the compiler flags
OBJ1 = file1.o utils.o      # object files for the first executable
OBJ2 = file2.o utils.o      # object files for the second executable

# the first target. Both executable will be made when 'make' is
# invoked with no target
all: prog1 prog2

# general rule how to compile a source file and produce an object file
.c.o:
	$(CC) $(CFLAGS) –c $<

# linking rule for the first executable
prog1: $(OBJ1)
	$(CC) $(CFLAGS) $(OBJ1) –o prog1

# linking rule for the second executable
prog2: $(OBJ2)
	$(CC) $(CFLAGS) $(OBJ2) –o prog2
depend:
	echo –e '\n' >> makefile
	$(CC) –MM *.c >> makefile
```

לאחר הפעלת 'make depend' מה – command-line יוספו חוקי התלות בסוף קובץ ה – makefile. פקודת ה – echo מוסיפה תו של ירידת שורה לפני חוקי התלות.

הגרסה האחרונה יכולה לשמש כשלד ל –makefileים שלכם בעתיד.

שימו לב שבכל פעם שנכתוב "make depend" ישורשרו ל – makefile מחדש כל החוקי התלות וחוקי התלות הישנים לא יימחקו. כאמור, כפילות בחוקי התלות אינה מהווה בעיה כך שאם

בפרוייקט שלנו רק נוספו תלויות מאז ההפעלה האחרונה של make depend , הכל יפעל כשורה. במקרה הנדיר יותר שבו נמחקה תלות בפרוייקט (אם מחקנו #include מסוים), הדבר לא יתבטא ב – makefile שלנו וייתכן שתתבצע קומפילציה מיותרת. ניתן כמובן לשנות את החוק ידנית או פשוט למחוק את כל חוקי התלות ולהריץ make depend מחדש.

השמה של macro מתוך ה – command-line

ניתן לקבוע את ערכם של ה – macro המוגדרים על ידינו מתוך ה – command-line. לדוגמה, אם נריץ את הפקודה:

```
$ make “CFLAGS = -Wall –g” prog1
```

הערך אותו השמנו ל- CFLAGS ב – makefile לא יהיה רלוונטי ובמקומו יקבל המקרו את הערך שניתן בפקודת ההרצה. הרצה כזאת של make תגרום לקומפילציה של prog1 עם אינפורמציה עבור ה – debugger.

ניתן גם להשתמש במקרו שלא הוצהר בקובץ ואז ערכו יהיה מחרוזת ריקה. לדוגמה:

```
.c.o:
	$(CC) $(CFLAGS) $(ADDFLAGS) –c $<
```

מכיוון ש – ADDFLAGS לא הוצהר כ – macro בשום מקום בקובץ, ערכו יהיה ריק וקיומו לא ישפיע על פקודת הקומפילציה. כאשר נריץ את הפקודה:

```
$ make “ADDFLAGS = -g”
```

אז קיומו ישפיע ויגרום לקומפילציה עם אינפורמציה ל – debugger.

Target שימושיים שנתן להוסיף

```
clean:
	rm –f *.o *~
```

בהפעלת

```
$ make clean
```

ימחקו (ללא בקשת אישור) כל קבצי ה – object וקבצים זמניים הנוצרים ע"י emacs .

```
backup:
	cp –f *.c *.h makefile ~/backup/
```

בהפעלת make backup יועתקו כל קבצי המקור בספרייה הנוכחית לספריית גיבוי (בהנחה שקיימת) . כל קובץ בעל אותו השם שכבר קיים ב – backup, יידרס (בגלל ה- ‘-f’)

build-in rules

נסו להשתמש בדוגמת ה makefile האחרונה, ולמחוק את שורת ההפקה הכללית:

```
.c.o:
        $(CC) $(CFLAGS) –c $<
```

למרבה הפלא make עדיין תבצע את הפעולות הדרושות למרות שפקודות הקומפילציה לא יכללו את הדגל Wall–. איך זה קרה ? מלבד החוקים שאנו מספקים ל make ב – makefile, התוכנה משתמשת במספר חוקים מובנים עבור קבצים בעלי סיומות מוכרות. מכיוון ש C היא שפה הנמצאת בשימוש רחב, מכיל make חוקים מובנים המטפלים בקומפילציה של קבצי c. החוקים הנכתבים ב makefile תמיד נמצאים בעדיפות על פני החוקים המובנים ולכן כאשר כתבנו חוק כללי משלנו לקומפילציה, הוא זה שפעל.

באופן כללי, אם בכל החוקים של target מסוים, לא מופיע אף פקודת הפקה, יש סיכוי שכלל מובנה (build-in) יפעל במטרה להפיק את ה – target.

Macro-ים המקבלים ערך באופן אוטומטי

$< - כפי שראינו – מקבל את שם הקובץ שגרם להפעלת החוק הכללי.

$* - מקרו המקבל את השם הרישא המשותפת לקובץ המפעיל ולקובץ המטרה של כלל כללי. לדוגמה:

```
.c.o:
        gcc -Wall –c $< -o $*.o
        echo $*.c "->" $*.o
```

יגרום להדפסה של "file1.c->file1.o" , לאחר הקומפילציה של file1.c .

$? - מקרו המקבל את רשימת כל הקבצים בחוק שתאריך השינוי שלהם מאוחר מתאריך השינוי של ה – target.

$@ מקרו המקבל את שם ה – target של החוק. לדוגמה:

```
check:
        echo $@
```

'make check' תגרום להדפסת 'check' על המסך.

פיצול שורה ארוכה בעזרת '\':

ניתן לפצל שורה ארוכה בעזרת התו '\'. לדוגמה:

```
file.o: file.c file1.h file2.h \
        file3.h file4.h \
        file5.h file6.h
        gcc -c file.c
```

make מתעלם מתו של ירידת שורה ('\n') ותווי רווח ו tab המופיעים לאחר '\'.

השקטה של make

במידה ולא מעוניינים ש make יכתוב כל פקודה שהוא מבצע על המסך, ניתן להריץ אותו באופן הבא:

```
> make –s
```

דרך נוספת היא להוסיף מטרה מדומה בקובץ ה- makefile:

```
.SILENT:
```

במידה ומעוניינים שרק פקודה מסוימת לא תודפס, מוסיפים את התו '@' בתחילתה:

```
backup:
        @ cp –f *.c *.h makefile ~/backup/
```

דגלים שימושיים של make

תחביר ההפעלה של make הוא כדלקמן:

```
make [ flags ] [ macro definitions ] [ targets ]
```

את שני הפרמטרים האחרונים כבר הכרנו, הפרמטר הראשון מורכב מדגלים המסמנים אפשרויות הפעלה. נתן מספר דגלים שימושיים:

make –n

הפעלת make ללא ביצוע פקודות ההפקה. לקלט הסטנדרטי יודפסו כל הפקודות שה- make היה מבצע בהפעלה רגילה אך, ללא ביצוען. גם פקודות מושתקות (ע"י @ או .SILENT) יודפסו.

make –r

הפעלה ללא שימוש בכללים המובנים

make –p

הדפסה של כל ערכי ה – macro המוגדרים וחוקי ההפקה.

make –i

התעלם מבעיות בביצוע פקודות ההפקה (error code שונה מאפס בהפעלת אחת הפקודות)

make –d

mode של debug. פרוט פעולות make. הרבה מלל, להשתמש כמוצא אחרון.

make –f

התייחס לקובץ אחר כקובץ ה – makefile. לדוגמה:

make –f mymakefile

או

make –f –

במקרה זה, כללי ה makefile יוכתבו ישירות מהמקלדת.

כלים אוטומטיים ליצירת makefile

מכיוון שתהליך כתיבת ה – makefile הוא מאד טכני, קיימות תוכנות המייצרות אותו באופן אוטומטי. לדוגמה, כאשר כותבים פרוייקט ב – visual studio, מיוצר ה-makefile ע"י תוכנה השייכת לסביבת העבודה, והמתכנת אינו צריך לדאוג לכך.
למרות שבדרך כלל, כלי אוטומטי ליצירת makefile הוא דבר מאד נוח, לעיתים הוא מגביל את המתכנת ולא מאפשר לו את הגמישות שהייתה לו כאשר כתב את ה – makefile בעצמו.
מלבד זאת, מכיוון שתוכנות ייצור ה – makefile צריכות לייצר קוד עבור פרוייקטים מסוגים שונים, עליהם להיות כתובות בצורה כללית, והקוד שהן מייצרות לפיכך הוא בדרך כלל מאד לא קריא (מיוצר מקרה פרטי של makefile כללי ומסובך).