[MAC0211] Laboratório de Programação I
Aula 15
GNU Make

Análise Léxica

Alair Pereira do Lago

DCC–IME–USP

21 de abril de 2015

[Aula passada] GNU Make

- É um utilitário que determina automaticamente quais pedaços de um grande programa precisam ser recompilados e dispara os comandos que os recompilam
- Pode ser usado com qualquer linguagem de programação cujo compilador possa ser executado com um comando *shell*
- Não se limita à construção de programas. Pode ser usado para descrever qualquer tarefa em que alguns arquivos precisam ser atualizados automaticamente a partir de outros sempre que houver alterações nesses outros

[Aula passada] *Makefile*

- para usar o *Make*, é preciso escrever um arquivo chamado *makefile*, que descreve os relacionamentos entre os arquivos do seu programa e indica comandos para atualizar cada arquivo. Exemplo: em um programa, o arquivo executável é construído a partir de arquivos objetos que, por sua vez, são gerados a partir de código-fonte. O *makefile* pode indicar como os fontes são compilados e ligados para gerar o executável.
- Uma vez que um *makefile* correto exista, basta executar o comando `make` para que todas as recompilações necessárias para a atualização do programa sejam executadas
- O *Make* usa as informações contidas no *makefile* e os horários da última modificação dos arquivos para decidir quando um arquivo precisa ser atualizado. O próprio *makefile* indica como a atualização deve ser feita

[Aula passada] Uma introdução a *Makefiles*

Regras

- Um *makefile* é composto por regras do tipo:

  alvo ... : pré-requisitos ...
  receita
  ...
  ...

- ***alvo***: geralmente é o nome de um arquivo que é gerado por um programa (ex.: arquivos executáveis ou objeto). Mas pode ser também o nome de uma ação a ser executada
- ***pré-requisito***: é um arquivo que é usado como entrada na criação do *alvo*. Um alvo pode depender de vários arquivos
- ***receita***: ação que o `make` executará. Pode possuir um ou mais comandos (na mesma linha ou um em cada linha). Importante: é preciso colocar um caractere de tab no início de cada linha da receita

[Aula passada] Uma introdução a *Makefiles*

Regras

- Em um regra, uma receita geralmente serve para criar o arquivo alvo quando há alteração em algum dos arquivos pré-requisitos.
- Mas uma regra não é obrigada a ter pré-requisitos (veremos um exemplo disso a seguir)
- Um makefile pode conter outras coisas além das regras. Mas um makefile simples só precisa delas para ser feito

[Aula passada] Um *makefile* simples

```
edit : main.o kbd.o command.o display.o \
            insert.o search.o files.o utils.o
       gcc -o edit main.o kbd.o command.o display.o \
            insert.o search.o files.o utils.o

main.o : main.c defs.h
       gcc -c main.c
kbd.o : kbd.c defs.h command.h
       gcc -c kbd.c
command.o : command.c defs.h command.h
       gcc -c command.c
display.o : display.c defs.h buffer.h
       gcc -c display.c
insert.o : insert.c defs.h buffer.h
       gcc -c insert.c
search.o : search.c defs.h buffer.h
       gcc -c search.c
files.o : files.c defs.h buffer.h command.h
       gcc -c files.c
utils.o : utils.c defs.h
       gcc -c utils.c
clean :
    rm edit main.o kbd.o command.o display.o insert.o search.o files.o utils.o
```

[Aula passada] Um *makefile* simples – funcionamento

- O caractere '\' é usado para dividir uma linha longa
- Para gerar o executável chamado `edit`, basta executar `make`
- Para apagar o arquivo executável e todos os arquivos objeto do diretório, executar `make clean`

Observações:

- Por padrão, o *Make* começa com o primeiro alvo do makefile. Ele é chamado de meta padrão (default goal). Metas são alvos que o *Make* se esforça para atualizar por último.
- Uma outra regra do *makefile* só é processada de forma automática no `make` se seu alvo aparece como pré-requisito para a meta ou se é pré-requisito para alguma outra regra da qual a meta depende

[Aula passada] *Makefile* – funcionamento

- Quando o alvo é um arquivo, ele precisa ser recompilado ou religado se um dos seus pré-requisitos mudam
- Um pré-requisito que é automaticamente gerado (ou seja, que é o alvo de uma regra também), deve ser atualizado antes de ser usado
- Por exemplo: `edit` depende de cada um dos 8 arquivos objeto. O objeto `main.o` depende do fonte `main.c` e o cabeçalho `def.h`
- O alvo `clean` não é um arquivo; é apenas o nome de uma ação. Ela não é pré-requisito para nenhuma outra regra e também não tem pré-requisitos. Por essa razão, a regra do `clean` só será executada pelo `make` quando isso for solicitado explicitamente.
- Alvos que são apenas ações (e não arquivos) são chamados de *alvos falsos* (*phony targets*)

[Aula passada] O uso de variáveis em *makefiles*

- *Variáveis* nos ajudam a evitar replicações no *makefile* e, consequentemente, diminuem a possibilidade de introdução de erros durante a manutenção
- Exemplo: a sequência de arquivos
  `main.o kbd.o command.o display.o insert.o search.o files.o utils.o`
  aparece mais de uma vez no *makefile* do editor de texto
- É uma prática frequente definir em um makefile uma variável de nome *OBJECTS*, *objs*, *OBJS*, *obj*, ou *OBJ* contendo a lista de todos os nomes dos arquivos objeto:

```
objects = main.o kbd.o command.o display.o insert.o \
                search.o files.o utils.o
```

[Aula passada] Um *makefile* simples (agora com variáveis)

```
objects = main.o kbd.o command.o display.o insert.o search.o files.o utils.o

edit : $(objects)
	gcc -o edit $(objects)

main.o : main.c defs.h
	gcc -c main.c
kbd.o : kbd.c defs.h command.h
	gcc -c kbd.c
command.o : command.c defs.h command.h
	gcc -c command.c
display.o : display.c defs.h buffer.h
	gcc -c display.c
insert.o : insert.c defs.h buffer.h
	gcc -c insert.c
search.o : search.c defs.h buffer.h
	gcc -c search.c
files.o : files.c defs.h buffer.h command.h
	gcc -c files.c
utils.o : utils.c defs.h
	gcc -c utils.c
clean :
    rm edit $(objects)
```

Regras implícitas

- Quando o alvo de uma regra é um arquivo com extensão `.o`, o *Make* é capaz de deduzir que ele deve usar o arquivo `.c` correspondente para gerar o alvo usando um comando 'gcc -c'. Por isso, podemos omitir as receitas para a geração de arquivos objeto
- Exemplo de regra que pode ser omitida no *makefile*

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

- Exemplo de regra válida no *makefile*

```
main.o :  def.h
```

- No exemplo acima, note que o arquivo `.c` usado para a geração do `.o` também é deduzido (ou seja, ele não precisa aparecer nos pré-requisitos da regra)
- Regras que omitem a receita são chamadas de *regras implícitas*

Um *makefile* simples (agora com regras implícitas)

```
objects = main.o kbd.o command.o display.o insert.o \
                search.o files.o utils.o

edit : $(objects)
       gcc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h
clean :
    rm edit $(objects)
```

Uma outra forma de definir *makefiles*

- Quando os objetos de um *makefile* são criados somente por regras implícitas, é possível usar um outro estilo de definição
- Nesse estilo de *makefile*, as entradas são agrupadas de acordo com seus pré-requisitos (e não por seus alvos)
- Exemplo:

```
objects = main.o kbd.o command.o display.o insert.o \
              search.o files.o utils.o

edit : $(objects)
     gcc -o edit $(objects)

$(objects) : defs.h
kbd.o command.o files.o : command.h
display.o insert.o search.o files.o : buffer.h
```

Regras falsas

- As regras falsas possibilitam a execução de comandos não relacionados à compilação
- Exemplo: remoção dos arquivos do programa

```
clean :
    rm edit $(objects)
```

- Como `clean` não é um pré-requisito para `edit`, sua regra nunca será executada com a chamada de `make` sem parâmetros. Ela só será executada quando for invocada explicitamente, com o comando `make clean`
- Um regra como a `clean` não deve aparecer como a primeira regra em um *makefile*, porque não queremos que seja a meta padrão

Regras falsas

- Considerando o exemplo da regra falsa `clean`, se um arquivo com o nome `clean` fosse criado no diretório do *makefile*, a receita de remoção dos arquivos do programa nunca mais seria executada. Como a regra falsa não tem pré-requisitos, o arquivo alvo criado "externamente" estaria sempre atualizado (o que faria com que a receita nunca mais fosse executada)
- Para se contornar esse problema, podemos declarar explicitamente que o alvo da regra é falso usando o alvo especial .PHONY. Exemplo:

```
.PHONY : clean
clean :
	-rm edit $(objects)
```

- No exemplo, usamos `-rm` em lugar do `rm`. O caracter '-' forçará que o `make` continue executando mesmo que a execução do `rm` retorne erros.

Erros em receitas

- Depois da execução no *shell* de uma linha de uma receita, o `make` olha para o status de saída. Se o *shell* completou com sucesso (status = 0), a próxima linha na receita é executada em um novo *shell*; depois que a última linha é terminada, a regra é terminada
- Se um erro ocorre na execução de um comando, o `make` abandona a regra atual e, possivelmente, as demais regras
- Em algumas situações, a falha em um comando da receita não precisa implicar na falha da regra toda. Por exemplo, na regra

```
main.o : main.c
	mkdir saida
	gcc -c main.c
```

  poderia não haver problema algum em continuar a receita em caso de falha no `mkdir`
- Colocar o caractere '-' no início do texto da linha (depois do tab) força o `make` a ignorar erros que ocorram nessa linha

Variáveis automáticas

- $< : armazena o nome do primeiro pré-requisito da regra
- $@ : armazena o nome do alvo
- $? : armazena a lista de nomes de todos os pré-requisitos mais novos que o alvo
- $^ : armazena a lista de nomes de todos os pré-requisitos

Exemplo de uso:

```
edit : main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
    gcc -o $@ $^
```

Geração automática de dependências

- Existem ferramentas que geram automaticamente regras de dependências entre arquivos e até mesmo *makefiles* completos
- Exemplos:
  - *gcc -M* e *gcc -MM* (`http://gcc.gnu.org/onlinedocs/gcc-4.1.2/gcc/Preprocessor-Options.html`)
  - *makedepend* (`http://manpages.ubuntu.com/manpages/hardy/man1/makedepend.1.html`)
  - *GNU Automake* (`http://www.gnu.org/software/automake/`)

Processamento de Linguagens de Programação

- **Sintaxe de uma linguagem**: especifica como os programas de uma linguagem são construídos
- **Semântica de uma linguagem**: especifica o quê os programas significam

Exemplo

Suponha que datas são construídas a partir de dígitos representados por *D* e o símbolo /, da seguinte maneira: *DD*/*DD*/*DDDD*

- De acordo com essa sintaxe, 01/02/2013 é uma data
- O dia a que essa data se refere não é identificado pela sintaxe. Por exemplo, no Brasil, essa data corresponde ao dia primeiro de fevereiro de 2013. Entretanto, nos EUA, ela corresponde ao dia dois de fevereiro de 2013
- Portanto, podemos ter mais de uma semântica associada à uma mesma sintaxe

Processamento de Linguagens de Programação

Etapas envolvidas em um processo de compilação:

- **Análise Léxica** – responsável por identificar os itens léxicos (palavras, números, símbolos, etc.) no código-fonte do programa
- **Análise Sintática** – responsável por identificar a estrutura sintática do programa e construir uma estrutura de dados (normalmente uma árvore sintática) para representar este programa em memória
- **Análise Semântica** – responsável por definir o que significa cada comando e gerar o código correspondente

Processamento de Linguagens de Programação

Neste curso abordaremos formalmente apenas a Análise Léxica

Processamento de programas é visto em:

- **MAC 316 – Conceitos de Linguagens de Programação** (construção de um interpretador)
- **MAC 414 – Linguagens Formais e Autômatos** (análise sintática em profundidade)
- **MAC 410 – Introdução à Compilação** (construção de um compilador)

Expressões e notações

- Expressões como $a + b * c$ são usadas há seculos e foram o ponto de início para o projeto de linguagens de programação
- Linguagens de programação usam um misto de notações. Por exemplo:
  - $a + b$ – operador aparece entre os operandos
  - $sqrt(a)$ – operador aparece antes do operando
  - $a++$ – operador aparece depois do operando
- **Notação infixa**: um operador binário é escrito entre os seus dois operandos (ex.: $a + b$)
- **Notação prefixa**: um operador binário é escrito antes dos seus dois operandos (ex.: $+ab$)
- **Notação pós-fixa**: um operador binário é escrito depois dos seus dois operandos (ex.: $ab+$)

Notação infixa

Vantagem dessa notação:

- É a mais "familiar" das notações e, por isso, mais fácil de se ler

Desvantagem dessa notação:

- Ambiguidade na decodificação. Exemplo: a expressão pode $a + b * c$ conduzir a duas decodificações diferentes:
  - a soma de a e $b * c$
  - a multiplicação de $a + b$ e c
- Desambiguação é feita com:
  - precedência de operadores
  - associatividade de operadores de mesma precedência
  - parênteses

Notação infixa

Exemplos:

- Operadores associativos-à-esquerda: +, -, * e /

  Correto: $4 - 2 - 1 = (4 - 2) - 1 = 2 - 1 = 1$

  Errado: $4 - 2 - 1 = 4 - (2 - 1) = 4 - 1 = 3$

- Operador associativo-à-direita: exponenciação

  $2^{3^4} = 2^{(3^4)} = 2^{81}$

Notação prefixa

Uma expressão em notação prefixa é escrita do seguinte modo:

- A notação prefixa de uma constante ou variável é a própria constate ou variável
- A aplicação de um operador **op** às sub-expressões E_1 e E_2 é escrita na notação prefix como **op** E_1 E_2

Vantagens dessa notação:

- É fácil de decodificar fazendo uma varredura da esquerda para a direita na expressão
- Dispensa o uso de parênteses (não há ambiguidade na identificação dos operandos de um operador)

Exemplos

- $* + 20\ 30\ 60 = * 50\ 60 = 3000$
- $* 20 + 30\ 60 = * 20\ 90 = 1800$

Notação pós-fixa (= *notação polonesa*)

Uma expressão em notação pós-fixa é escrita do seguinte modo

- A notação pós-fixa de uma constante ou variável é a própria constate ou variável
- A aplicação de um operador **op** às sub-expressões E_1 e E_2 é escrita na notação pós-fixa como E_1 E_2 **op**

Vantagens dessa notação:

- São fáceis de se avaliar de forma automatizada com o auxílio de uma estrutura de pilha
- Dispensa o uso de parênteses (não há ambiguidade na identificação dos operandos de um operador)

Exemplos

- 20 30 + 60 * = 50 60 * = 3000
- 20 30 60 + * = 20 90 * = 1800

Notação "*mixfixa*"

Operações especificadas por uma combinação de símbolos não se enquadram na classificação infixa, prefixa e pós-fixa.

Exemplo

$$\textbf{if } a > b \textbf{ then } a \textbf{ else } b$$

Análise Léxica (na prática)

Exemplo: Calculadora HP

- **Notação pós-fixa** (também chamada de **notação polonesa**)
- Ver código da implementação disponível no Paca

Bibliografia e materiais recomendados

- Manual do GNU Make
  `http://www.gnu.org/software/make/manual/`
- Capítulo 2 do livro *Programming Languages – Concepts & Constructs*, de Ravi Sethi
- Notas das aulas de MAC0211 de 2010, feitas pelo Prof. Kon
  `http://www.ime.usp.br/~kon/MAC211`

Materiais recomendados sobre Latex

- Página do projeto: `http://www.latex-project.org/`
- Instalação no Linux: pacote `texlive-latex-base`
- Editores para Latex:
  - *Texmaker* (pacote *texmaker*), *TeXstudio* (pacote *texstudio*), *Kile* (pacote *kile*)
  - Tabela comparativa: `http://en.wikipedia.org/wiki/Comparison_of_TeX_editors`
- *A Not so short introduction to Latex*: `http://mirrors.ctan.org/info/lshort/english/lshort.pdf` Versão em português de Portugal: `http://mirrors.ctan.org/info/lshort/portuguese/pt-lshort.pdf`
- *Wiki Guide*: `http://en.wikibooks.org/wiki/LaTeX`
- Um guia bem curto para iniciantes: `https://www.cs.princeton.edu/courses/archive/spring10/cos433/Latex/latex-guide.pdf`

Cenas dos próximos capítulos...

Na próxima aula:

- Revisão para a prova
- Mais sobre análise léxica
- Processamento de macros (?)