CS 536 Announcements for Tuesday, February 15, 2022

Programming Assignment 2
- due Friday, February 25

Homework 1 is available

Last Time
- why regular expressions aren't enough
- CFGs
  - formal definition
  - examples
  - language defined by a CFG
- parse trees

Today
- Makefiles
- list grammars
- resolving ambiguity

Next Time
- wrap up CFGs
- syntax-directed translation

CFG review

formal definition: CFG $G = (N, \Sigma, P, S)$

CFG generates a string by applying productions until no non-terminals remain

$\Rightarrow^+ \text{ means "derives in 1 or more steps"}$

language defined by a CFG $G$

$L(G) = \{ w | s \Rightarrow^+ w \}$ where
- $s$ = start is the start non-terminal of $G$, an
- $w$ = sequence consisting of (only) terminal symbols or $\varepsilon$

Example: nested parentheses

- $N = \{ q, 3 \}$
- $\Sigma = \{ , (, ) \}$
- $P = q \Rightarrow (q)$
- $S = q$
- $q \Rightarrow \varepsilon$
- $q \Rightarrow (q)$
- $q \Rightarrow (\varepsilon)$
- $q \Rightarrow + (\varepsilon)$
Makefiles

Basic structure

\(<\text{target}>\): \(<\text{dependency list}>\)
\(<\text{command to satisfy target}>\)

Example

Example.class: Example.java IO.class
  javac Example.java

IO.class: IO.java
  javac IO.java

Make creates an internal dependency graph
- a file is rebuilt if one of its dependencies changes

Variables – for common configuration values to use throughout your makefile

Example

JC = /s/std/bin/javac
JFLAGS = -g  \(\text{\textless build for use with debugger\textgreater}\)

Example.class: Example.java IO.class
  $(JC) $(JFLAGS) Example.java

IO.class: IO.java
  $(JC) $(JFLAGS) IO.java

Phony targets
- target with no dependencies = “phony”
- used make to run commands:

Example

\textbf{clean:}
  \texttt{rm -f *.class}

\textbf{test:}
  \texttt{java Example infile1.txt outfile1.txt}
  \texttt{java Example infile2.txt outfile1.txt}
Programming Assignment 2

Modify:
- minim.jlex
- P2.java
- Makefile

Makefile

```bash
###
# testing - add more here to run your tester and compare
# its results to expected results
###
test:
    java -cp $(CP) P2
diff allTokens.in allTokens.out
###
# clean up
###
clean:
    rm -f *~ *.class minim.jlex.java

cleantest:
    rm -f allTokens.out
```

Running the tester

```
royal-30(51)% make test
java -cp ./deps:. P2
3:1 ****ERROR**** ignoring illegal character: a
diff allTokens.in allTokens.out
3d2 < a
make: *** [Makefile:40: test] Error 1
```
Derivation order

1) prog → BEGIN stmts END
2) stmts → stmts SEMICOLON stmt
3)     | stmt
4) stmt → ID ASSIGN expr
5) expr → ID
6)     | expr PLUS ID

Leftmost derivation: leftmost non-terminal is always expanded

Rightmost derivation: rightmost non-terminal is always expanded
Expression Grammar Example

1) \( \text{expr} \rightarrow \text{INTLIT} \)
2) \( \mid \text{expr PLUS expr} \)
3) \( \mid \text{expr TIMES expr} \)
4) \( \mid \text{LPAREN expr RPAREN} \)

Derive: \( 4 + 7 \times 3 \)

For grammar \( G \) and string \( w \), \( G \) is **ambiguous** if there is

\( \geq 1 \) leftmost derivation of \( w \) OR \( \geq 1 \) rightmost derivation of \( w \) OR \( \geq 1 \) parse tree for \( w \)

Grammar \( G \)/ \( \geq 1 \) leftmost derivation of \( D \)

\[ a \rightarrow b \mid c \]
\[ b \rightarrow D \]
\[ c \rightarrow D \]

\( a \rightarrow a \)
\( b \rightarrow b \)
\( c \rightarrow c \)
\( D \rightarrow D D \)
Grammars for expressions

Goal: write a grammar that correctly reflects precedences and associativities

- use different non-terminal for each precedence level
- start by re-writing production for lowest precedence operator first

Example

1) expr → INTLIT
2) | expr PLUS expr
3) | expr TIMES expr
4) | LPAREN expr RPAREN

Try to get * evaluated last

```
expr → expr PLUS expr
| term

term → term TIMES term
| factor

factor → INTLIT
| LPAREN expr RPAREN
```

```
4 + 7 * 3
  
expr
  +
  
  
  term
  TIMES
  
  
  term
  TIMES
  
  term
  LPAREN
  expr
  RPAREN
```

Can't get to PLUS unless add parens

```
INTLIT (4)
INTLIT (7)
INTLIT (3)
```
Grammars for expressions (cont.)

What about associativity? Consider $1 + 2 + 3$

Definition: recursion in grammars

A grammar is **recursive** in non-terminal $x$ if $x \Rightarrow + \alpha \times y$ for non-empty strings of symbols $\alpha$ and $y$

A grammar is **left-recursive** in non-terminal $x$ if $x \Rightarrow + x \gamma$ for non-empty string of symbols $\gamma$

A grammar is **right-recursive** in non-terminal $x$ if $x \Rightarrow + \alpha x$ for non-empty string of symbols $\alpha$

In expression grammars

for left associativity, use left recursion
for right associativity, use right recursion

Example

In expression grammars

for left associativity, use left recursion
for right associativity, use right recursion

```
expr -> expr PLUS expr
    | term

term -> term TIMES factor
    | 1

factor -> INTLIT
    | LPAREN expr RPAREN
```

Example

- Expression: $1 + 2 + 3$
- Parse tree:
  - $1 + 2 + 3$
  - $\Rightarrow (1 + 2) + 3$
  - $\Rightarrow 3 + 3$
  - $\Rightarrow 6$

Note: In the expression grammar, the associativity is determined by the placement of parentheses.
You try: add exponentiation (POW)

Add exponentiation (POW) to this grammar, with the correct precedence and associativity.

\[
\begin{align*}
\text{expr} & \rightarrow \text{expr PLUS term} \\
& \quad \mid \text{term} \\
\text{term} & \rightarrow \text{term TIMES factor} \\
& \quad \mid \text{factor} \\
\text{factor} & \rightarrow \text{INTLIT} \\
& \quad \mid \text{LPAREN expr RPAREN}
\end{align*}
\]