CS536 Lecture 8

Thursday 12 February 2015

Reminders:

- Reading

Last class:

- Makefiles
- Ambiguity in grammars and resolving it

Today:

- Fixing Associativity in an ambiguous grammar
- Syntax-Directed Translation

Ambiguity in Grammars

Resolving Ambiguity 1: Precedence

Intuitive Problem:

Fix precedence by

- Having a different nonterminal for each *level* of precedence
- Expanding lower precedence operators *higher* in the parse tree

Instead of

```
Expr → INTLIT
     | Expr MINUS Expr
     | Expr TIMES Expr
     | LPAREN Expr RPAREN
```

we have

```
Expr → Expr MINUS Expr
     | Term

Term → Term TIMES Term
     | Factor

Factor → INTLIT
       | LPAREN Expr RPAREN
```

Example Derivation:

```
4 - 7 * 3
```

Second problem: Ambiguity in Associativity

Consider `4 - 7 - 3`

Grammar:

```
Expr → Expr MINUS Expr
     | Term

Term → Term TIMES Term
     | Factor

Factor → INTLIT
       | LPAREN Expr RPAREN
```

Parse tree(s):

Resolving Ambiguity 2: Associativity

Where is the ambiguity coming from?

Recursion in grammars:

A grammar is *recursive* in nonterminal X if:

$X \Rightarrow^+ \alpha X \beta$ for non-empty strings of symbols α and β

A grammar is *left-recursive* in nonterminal X if:

$X \Rightarrow^+ X \beta$ for non-empty string of symbols β

A grammar is *right-recursive* in nonterminal X if:

$X \Rightarrow^+ \alpha X$ for non-empty string of symbols α

Solution:

For left-associativity, we'll use _____.

For right-associativity, we'll use _____.

Resolving Ambiguity 2: Associativity (cont'd)

Fixed grammar:

Expr → *Expr* MINUS ~~*Expr*~~ *Term*
| *Term*

Term → *Term* TIMES ~~*Term*~~ *Factor*
| *Factor*

Factor → INTLIT
| LPAREN *Expr* RPAREN

Derive `4 - 7 - 3`

On Your Own

Add exponentiation to the above grammar with the right precedence and associativity. Recall exponentiation is right-associative and higher in precedence than +,-,*, and /.

Hint: Add another nonterminal symbol.

Syntax-Directed Translation

CFGs for definition vs. recognition

Translating a sequence of tokens

Augment CFG with translation rules. A LHS nonterminal is a function of:

- constants
- RHS nonterminal translations
- RHS terminal value

SDT Example 1

Computing the decimal value of binary representations:

CFG:	Rules:
$B \rightarrow$ **0**	B.trans = 0
| **1**	B.trans = 1
| B**0**	B.trans = B_2.trans*2
| B**1**	B.trans = B_2.trans*2 + 1

Trace example string: 10110

Parse tree:

SDT Example 2: Variable Declarations

Translating a string of declarations:

CFG:		Rules:
DeclList	→ ϵ	*DeclList*.trans = “”
	| *Decl DeclList*	*DeclList.trans* = *Decl*.trans + “ “ + $DeclList_2$.trans
Decl	→ *Type* ID ;	*Decl*.trans = ID.value
Type	→ INT	
	| BOOL	

Trace example:

```
int xx;
bool yy;
```

Variation: Only add `int` declarations to the output string.

Abstract Syntax Trees

We can translate a sequence of tokens into an Abstract Syntax Tree (parsing).

Abstract Syntax Tree: A condensed form of the parse tree.

Example: `(5+2)*8`

From our earlier grammar, the parse tree is:

ASTs for Parsing

Why AST?

How do we build it?

Implementing ASTs:

How do we represent trees in code?