COMP 520 - Compilers

Lecture 7 (Thu Feb 3, 2022)

Operator Precedence and Stratified Grammars

- wa1 and wa2 sample solutions are online
- pa1 is due tonight by midnight

Topics

- Expressing operator precedence using stratified grammars
  - Grammar structure
  - LL(1) parsing

- Constructing corresponding syntax trees
  - using recursive descent parsers

The shape of the syntax tree

- Intuition
  - bottom up evaluation of expressions in AST
  - therefore nodes lower in the tree are evaluated before their parents

- Associativity and precedence in arithmetic expressions

  2 + 3 + 4
  - left to right evaluation => left associativity
  - tree is deep on the left

  - - - 3
  - right to left evaluation of unary op => right associative
  - tree is deep on the right

  2 + 3 * 4
  - operator precedence
  - tree is deep on right since * has higher precedence than +

  (2 + 3) * 4
  - explicit precedence
  - tree is deep on the left

Specifying operator precedence in an LL(1) grammar

- Suppose we have a simple grammar to describe arithmetic expressions

  E ::= E + E | E * E | (E) | num

- Consider the string of terminals 2+3*4
  - the string has two syntax trees
    - the grammar is ambiguous
  - one of these trees reflects the desired operator precedence
    - multiplication should be “lower” in the tree than addition
      - interpretation: must evaluate multiplication before we can evaluate addition
  - How can we encode precedence in the grammar?

Simple unambiguous grammar for expressions

- Our familiar grammar for arithmetic expressions

  E ::= T | E Op T

  T ::= (E) | num

  Op ::= + | *

- What is the associativity?

- Does it enforce precedence?

- What is the shape of the syntax tree of the following?
  - 2 + 3 + 4
  - 2 + (3 + 4)
  - (2 + 3) + 4

- Is this grammar LL(1)?

Incorporating precedence in expressions

- Operator associativity and precedence can be specified using a *stratified* grammar

```
E ::= E + T | T
T ::= T * F | F
F ::= (E) | num
```

- Associativity: consider the sentence 2+3+4
  - what is the shape of the syntax tree?
- Precedence: consider the sentences 2+3*4 and 2*3+4
  - why does it work ?
- Exercise: construct the syntax tree for 3+4*5+6

Parsing stratified grammar

- Stratified grammar has left recursion

  E ::= E + T | T

  T ::= T * F | F

  F ::= (E) | num

- Eliminate left recursion

  E ::= T (+ T)*

  T ::= F (* F)*

  F ::= (E) | num

- Augment grammar
  - add unique start symbol S and terminal $ representing end-of-input

    S ::= E $

Recursive-descent parsing of stratified grammar

- Stratified grammar in EBNF form

  S ::= E $ (1)

  E ::= T (+ T)* (2)

  T ::= F (* F)* (3)

  F ::= (E) | num (4)

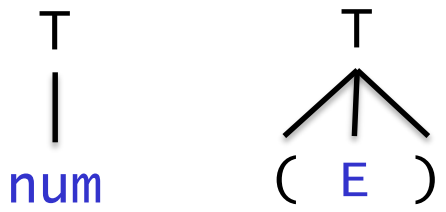
- Is it LL(1)?

How can we build an *abstract* syntax tree?

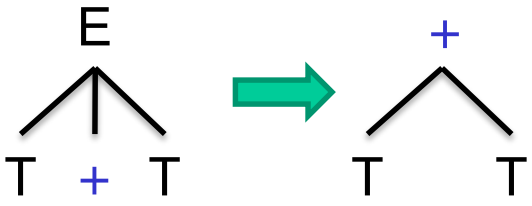
- Idea
  - Each parse method returns a syntax tree
  - Syntax tree is built bottom-up
  - Ex:

```
E ::= T + T
T ::= ( E ) | num
```

  - `parseT()`
    - returns a num leaf or
    - returns an E tree



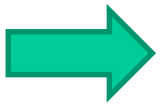
  - `parseE()`
    - returns a T + T ternary tree



How can this work with grammar transformations?

- Left recursion removal

E ::= T | E op T
T ::= (E) | num



E ::= T (op T)*
T ::= (E) | num

```
ExprTree parseE() {
    ExprTree e1 = parseT();
    while (curToken.kind == Token.op) {
        String op = curToken.spelling;
        acceptIt();
        ExprTree e2 = parseT();
        e1 = new ExprTree(e1,op,e2);
    }
    return e1;
}
```


PA2 abstract syntax tree construction

- PA2 abstract syntax tree constructors