

Grammars and Parsing

Application of recursion

- So far, we have written recursive programs on integers.
- Let us now consider a new application, grammars and parsing, that shows off the full power of recursion.
- Parsing has numerous applications: compilers, data retrieval, data mining,....

Grammars

Sentence → Noun Verb Noun
Noun → boys
Noun → girls
Noun → dogs
Verb → like
Verb → see

- **Grammar:** set of rules for generating sentences in a language.
- Our sample grammar has these rules:
 - a Sentence can be a Noun followed by a Verb followed by a Noun
 - a Noun can be 'boys' or 'girls' or 'dogs'
 - a Verb can be 'like' or 'see'
- Examples of Sentence:
 - boys see dogs
 - dogs like girls
 -
- Note: white space between words does not matter
- This is a very boring grammar because the set of Sentences is finite (exactly 18 sentences). Work this out as an exercise.

Recursive grammar

Sentence → Sentence and Sentence
Sentence → Sentence or Sentence
Sentence → Noun Verb Noun
Noun → boys
Noun → girls
Noun → dogs
Verb → like
Verb → see

- Examples of Sentences in this language:
 - boys like girls
 - boys like girls and girls like dogs
 - boys like girls and girls like dogs and girls like dogs
 - boys like girls and girls like dogs and girls like dogs and girls like dogs
 -
- This grammar is more interesting than the one in the last slide because the set of Sentences is infinite.
- What makes this set infinite? Answer: recursive definition of Sentence

Detour

- What if we want to add a period at the end of every sentence?
- Does this work?

Sentence \rightarrow Sentence and Sentence .

Sentence \rightarrow Sentence or Sentence .

Sentence \rightarrow Noun Verb Noun .

Noun \rightarrow

No! This produces sentences like
girls like boys . and boys like dogs .

Sentences with periods

TopLevelSentence \rightarrow Sentence .

Sentence \rightarrow Sentence and Sentence

Sentence \rightarrow Sentence or Sentence

Sentence \rightarrow Noun Verb Noun

Noun \rightarrow boys

Noun \rightarrow girls

Noun \rightarrow dogs

Verb \rightarrow like

Verb \rightarrow see

- Add a new rule that adds a period only at the end of the sentence.
- Thought exercise: how does this work?
- End of detour

Grammar for simple expressions

Expression \rightarrow integer

Expression \rightarrow (Expression + Expression)

- This is a grammar for simple expressions:
 - An E can be an integer.
 - An E can be '(' followed by an E followed by '+' followed by an E followed by ')'
- Set of Expressions defined by this grammar is a recursively-defined set.

E \rightarrow integer

E \rightarrow (E + E)

Here are some legal expressions:

2

(3 + 34)

((4+23) + 89)

((89 + 23) + (23 + (34+12)))

Here are some illegal expressions:

(3

3 + 4

Parsing

- **Parsing:** given a grammar and some text, determine if that text is a legal sentence in the language defined by that grammar
- For many grammars such the simple expression grammar, we can write efficient programs to answer this question.
- Next slides: parser for our small expression language
 - Caveat: code uses CS211In object for doing input from a file, so it is not an Ur-Java program.
 - However, you should understand the structure of the code to see the parallel between the language definition (recursive set) and the parser (recursive function)

Helper class: CS211In

- Read the on-line code for the CS211In class
- Code lets you
 - open file for input:
 - CS211In f = new CS211In(String-for-file-name)
 - examine what the next thing in file is: f.peekAtKind()
 - Integer?: such as 3, -34, 46
 - Word?: such as x, r45, y78z (variable name in Java)
 - Operator?: such as +, -, *, (,), etc.
 - read next thing from file:
 - integer: f.getInt()
 - Word: f.getWord()
 - Operator: f.getOp()

- Useful methods in CS211In class:

- f.check(char c):
 - Example: f.check('*'); //true if next thing in input is *
 - Check if next thing in input is c
 - If so, eat it up and return true
 - Otherwise, return false
- f.check(String s):
 - Example of its use: f.check("if");
 - Return true if next thing in input is word if

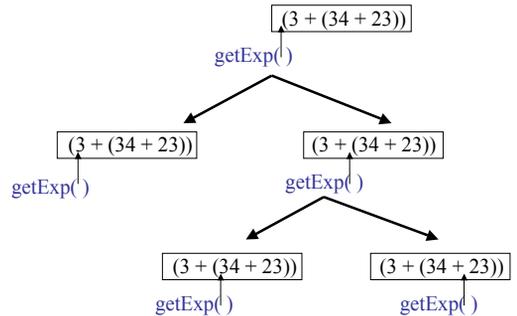
Parser for expression language

```
static boolean expParser(String fileName) //returns true if file has single expression
CS211In f = new CS211In(fileName);
boolean gotIt = getExp(f);
if (f.peekAtKind() == CS211In.EOF) //no junk in file after expression
    return gotIt;
else //file contains some junk after expression, so return false
    return false;
}
static boolean getExp(CS211In f) //reads one expression from file
switch (f.peekAtKind()) {
case CS211In.INTEGER: //E → integer
    {f.getInt();
    return true;
}
case CS211In.OPERATOR: //E → (E+E)
    return f.check('(') && getExp(f) && f.check('+') &&
        getExp(f) && f.check(')');
default: return false;
}
}
```

Note on boolean operators

- Java supports two kinds of boolean operators:
 - E1 & E2:
 - Evaluate both E1 and E2 and compute their conjunction (i.e., "and")
 - E1 && E2:
 - Evaluate E1. If E1 is false, E2 is not evaluated, and value of expression is false. If E1 is true, E2 is evaluated, and value of expression is the conjunction of the values of E1 and E2.
- In our parser code, we use &&
 - if "f.check('(') returns false, we simply return false without trying to read anything more from input file. This gives a graceful way to handling errors.
 - don't worry about this detail if it seems too abstruse...

Tracing recursive calls to getExp



Modifying parser to do SaM code generation

- Let us modify the parser so that it generates SaM code to evaluate arithmetic expressions: (eg)

```
2           : PUSHIMM 2
            STOP
(2 + 3)     : PUSHIMM 2
            PUSHIMM 3
            ADD
            STOP
```

Idea

- Recursive method getExp should return a string containing SaM code for expression it has parsed.
- Top-level method expParser should tack on a STOP command after code it receives from getExp.
- Method getExp generates code in a recursive way:
 - For integer i, it returns string "PUSHIMM" + i + "\n"
 - For (E1 + E2),
 - recursive calls return code for E1 and E2
 - say these are strings S1 and S2
 - method returns S1 + S2 + "ADD\n"

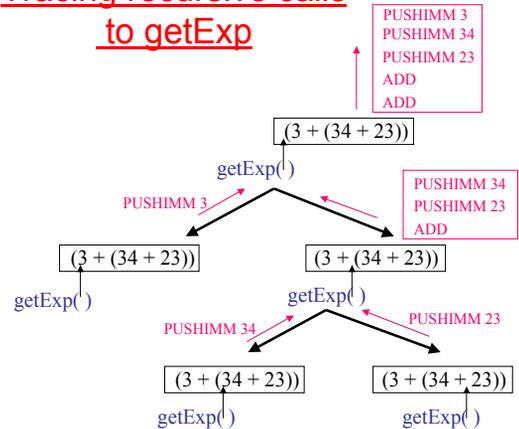
CodeGen for expression language

```

static String expCodeGen(String fileName) //returns SaM code for expression in file
    CS211In f = new CS211In(fileName);
    String pgm = getExp(f);
    return pgm + "STOP\n"; //not doing error checking to keep it simple
}
static String getExp(CS211In f) //no error checking to keep it simple
    switch (f.peekAtKind()) {
    case CS211In.INTEGER: //E → integer
        return "PUSHIMM" + f.getInt() + "\n";
    case CS211In.OPERATOR: //E → (E+E)
        {
            f.check('(');
            String s1 = getExp(f);
            f.check('+');
            String s2 = getExp(f);
            f.check(')');
            return s1 + s2 + "ADD\n";
        }
    default: return "ERROR\n";
    }
}

```

Tracing recursive calls to getExp



Exercises

- Think about recursive calls made to parse and generate code for simple expressions
 - 2
 - (2 + 3)
 - ((2 + 45) + (34 + -9))
- Can you derive an expression for the total number of calls made to getExp for parsing an expression?
 - Hint: think inductively
- Can you derive an expression for the maximum number of recursive calls that are active at any time during the parsing of an expression?

Number of recursive calls

- Claim:
 - # of calls to getExp for expression E =
 - # of integers in E +
 - # of addition symbols in E.
- Example: ((2 + 3) + 5)
- # of calls to getExp = 3 + 2 = 5

Conclusion

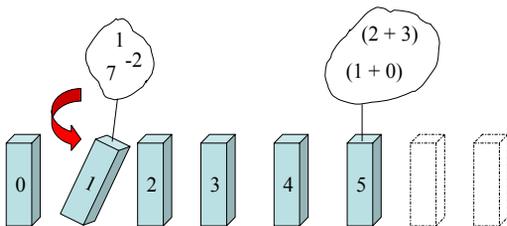
- Recursion is a very powerful technique for writing programs.
- Common mistakes:
 - Unwinding the recursion mentally (where do you stop??)
 - Incorrect or missing base cases
- Try to write “mathematical” description of the recursive algorithm like we have been doing, and reason about base cases etc. before writing program.
 - Why? Syntactic junk such as type declarations etc. may create mental fog which obscures the underlying recursive algorithm.

Extra material: Number of recursive calls

- Claim:
of calls to getExp for expression E =
of integers in E +
of addition symbols in E.
Example: $((2 + 3) + 5)$
of calls to getExp = $3 + 2 = 5$

Inductive Proof

- Order expressions by their length (# of tokens)
- $E1 < E2$ if $\text{length}(E1) < \text{length}(E2)$.



Proof of # of recursive calls

- *Base case:* (length = 1) Expression must be an integer. getExp will be called exactly once as predicted by formula.
- *Inductive case:* Assume formula is true for all expressions with n or fewer tokens.
 - If there are no expressions with $n+1$ tokens, result is trivially true for $n+1$.
 - Otherwise, consider expression E of length $n+1$. E cannot be an integer; therefore it must be of the form $(E1 + E2)$ where E1 and E2 have n or fewer tokens. By inductive assumption, result is true for E1 and E2. (contd. on next slide)

Proof(contd.)

#-of-calls-for-E =
= 1 + #-of-calls-for-E1 + #-of-calls-for-E2
= 1 + #-of-integers-in-E1 + #-of-'+'-in-E1 + #-
of-integers-in-E2 + #-of-'+'-in-E2
= #-of-integers-in-E + #-of-'+'-in-E
as required.