DD1362
Programming Paradigms

Formal Languages and Syntactic Analysis
Lecture 3

Philipp Haller

April 19th, 2021

Review of Lecture 2

- Finite automata formally
- Regular languages
  - A class of formal languages that can be described ***using regular expressions or finite automata***
  - Regular expressions and finite automata have the ***same expressive power***
- Context-free grammars
  - Express strictly more languages than regex

Finite automaton = 5-tuple of different sets

Example: $L = \{ a^n b^n \mid n >= 0 \}$

Today's Lecture

- Lexical analysis
- Derivations and parse trees
- Recursive descent parsing
- Eliminating ambiguity

Lexical Analysis

Lexical Analysis

Lexical analysis: The process of transforming a sequence of (individual) characters into a sequence of ***tokens***

irrelevant ***for parsing***

Goals:

1. ***Remove irrelevant parts*** of input string, for example:
   - whitespace (spaces, newlines, tabs, …)
     Input `"12+5"` should be treated the same as `"12 + 5"`
   - code comments (do not affect executable binaries)
2. ***Abstract away details*** from grammar, for example:
   Longest match rule:
   `"hello123 45"` should be treated as the token sequence
   `Ident(hello123) Num(45)` rather than
   `Ident(hello) Num(123) Num(4) Num(5)`

Lexical Analysis

Lexical analysis: The process of transforming a sequence of (individual) characters into a sequence of ***tokens***

irrelevant ***for parsing***

Goals:

1. ***Remove irrelevant parts*** of input string, for example:
   - whitespace (spaces, newlines, tabs, …)
     Input `"12+5"` should be treated the same as `"12 + 5"`

2.

Rule of thumb: if a part of the language can be described using a simple regular expression then it is usually better to consider it as a kind of token.

Lexical Analysis of Numbers

Idea: pre-process input string such that numbers are represented as complete tokens

Example: consider the string `"378*232*(582-01)"`

- Input string is equal to the character sequence
  `` `3',`7',`8',`*',`2',`3',`2',`*',`(',`5',`8',`2',`-',`0',`1',`)' ``
- Lexical analysis transforms this sequence into a new sequence of ***tokens***
  `Num, '*', Num, '*', '(', Num, '-', Num, ')'`
- Some tokens correspond to single characters (like `'*'` or `'('`), others consist of entire substrings (like `Num`)
- Tokens may also carry ***token data*** like the integer value of a number, for example `Num(378)`
- The generated sequence of tokens is the ***input to the parser***

Lexical Analysis: Example

Consider the following Haskell function:

```
-- This here is my function
myFunction alpha beta =
5 * x
where
-- compute difference
x = alpha-beta
```

What tokens do we have?
Keywords: where, let, etc.
Operators and symbols: Plus, Minus, Times, Equal, etc.
Name: name of variable/function/etc.
... and many more

Lexical Analysis: Example

Consider the following Haskell function:

```
-- This here is my function
myFunction alpha beta =
5 * x
where
-- compute difference
x = alpha-beta
```

Possible tokenization:

Name, Name, Name, Equal, Int, Times, Name, Where, Name, Equal, Name, Minus, Name

Derivations and Parse Trees

Derivation

Expr → Num
| Expr + Expr
| Expr - Expr
| Expr * Expr
| Expr / Expr
| (Expr)

A ***derivation*** of an input string is a sequence of grammar rules that are applied to produce that string.

Example: Let us derive "4*(5+3)", i.e., the token sequence Num, '*', '(', Num, '+', Num, ')'

1. Start with ***start symbol*** of grammar
2. Each step: ***replace exactly one non-terminal symbol*** with the right-hand side of one of its productions

<u>Expr</u> → <u>Expr</u> * Expr → Num * <u>Expr</u> → Num * (<u>Expr</u>) → Num * (<u>Expr</u> + Expr) → Num * (Num + <u>Expr</u>) → Num * (Num + Num)

Parse Tree

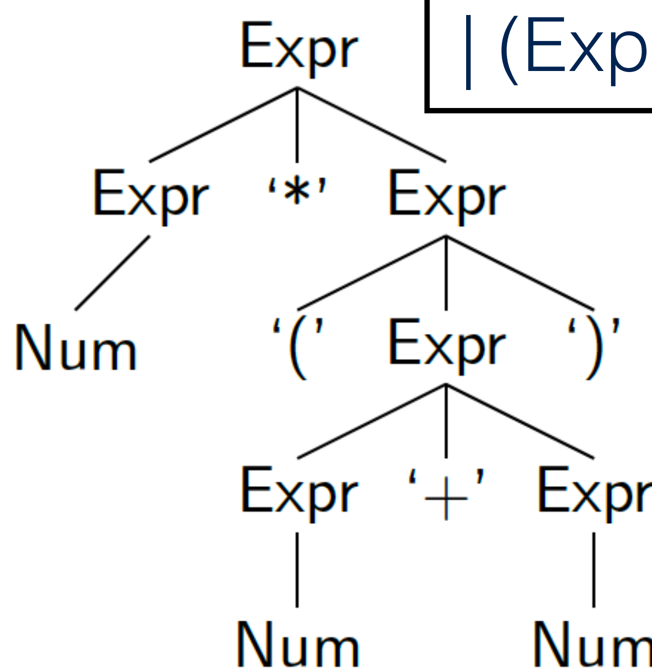
A ***parse tree*** represents a (set of) derivation(s) and encapsulates the ***semantics*** of an input string

Expr → Num
| Expr + Expr
| Expr - Expr
| Expr * Expr
| Expr / Expr
| (Expr)

Derivation:

Expr → Expr * Expr → Num * Expr → Num * (Expr) → Num * (Expr + Expr) → Num * (Num + Expr) → Num * (Num + Num)

Parse tree:

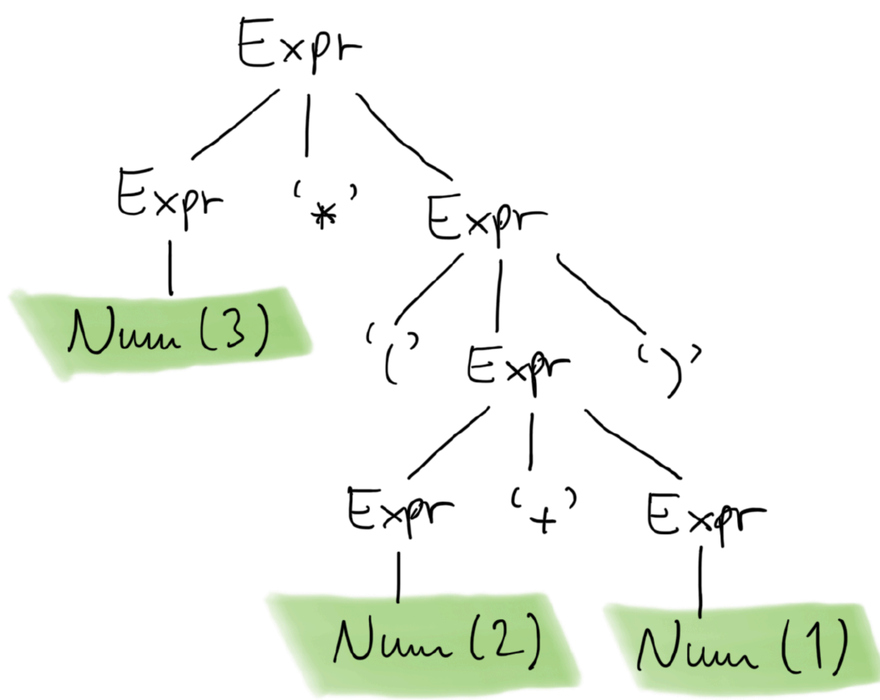


Parse Tree: Semantics

The parse tree encapsulates the semantics of an expression/a program.

- Parse tree enables ***evaluating an expression***: “execute” computation to obtain result ***value***
- Need to know the actual integers of the Num tokens → add this as ***token data***
- Evaluation done using ***tree traversal***

Parse tree:



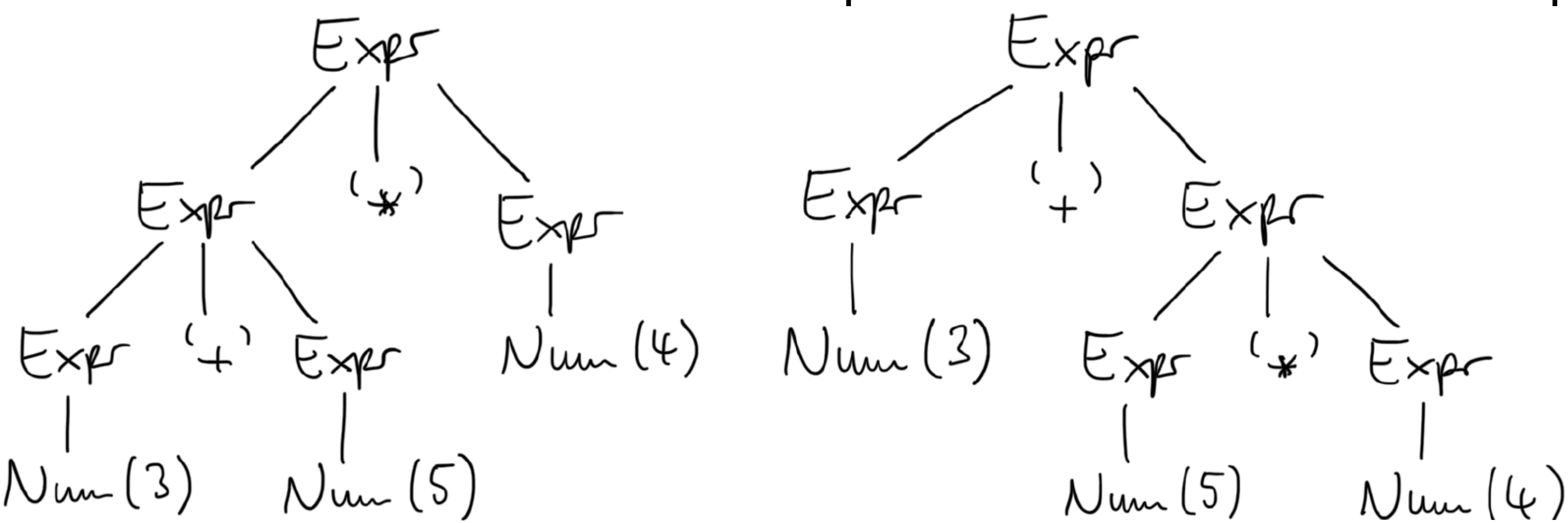
Derivations & Trees

Consider the expression 3 + 5 * 4

- Can you find two different derivations?
- What about two different parse trees?

Solution:

Expr → Num



Derivations

Expr → Num
| Expr + Expr | Expr - Expr
| Expr * Expr | Expr / Expr
| (Expr)

Consider the expression 3 + 5 * 4

- Can you find two different deriva
- What about two different parse trees?

Solution:

Both are correct parse trees of the given expression according to the grammar!

This means there are two meanings!

The grammar is ***ambiguous***.

Recursive Descent Parsing

Recursive Descent Parsing

A method for constructing an efficient parser for a given grammar:

- An input string is parsed according to the productions needed for its derivation
- When starting to parse a (string derived from a) non-terminal, ***look ahead*** to the next token to select the corresponding production
- For each non-terminal symbol, create a ***recursive function*** responsible for parsing that non-terminal
- Parse according to a production as follows:
  - For each terminal: check that it matches the next token
  - For each non-terminal symbol: call the function corresponding to the non-terminal symbol

Grammar for Binary Trees

```
<BinTree> ::= Leaf LPar Num RPar
    | Branch LPar <BinTree> Comma <BinTree> RPar
```

Terminal symbols:

- Leaf: `"leaf"`
- Branch: `"branch"`
- Num: `[0-9]+`
- LPar, RPar, Comma: parentheses and comma

Example:

```
branch(branch(leaf(17),leaf(42)),leaf(5))
```

Result of lexical analysis:

```
Branch LPar Branch LPar Leaf LPar Num RPar Comma Leaf
   LPar Num RPar RPar Comma Leaf LPar Num RPar RPar
```

```
<BinTree> ::= Leaf LPar Num RPar
  | Branch LPar <BinTree> Comma <BinTree> RPar
```

Recursive function BinTree for parsing <BinTree>:

```
ParseTree BinTree() throws SyntaxError {
  Token t = lexer.peekToken();
  if (t.getType() == TokenType.Leaf) {
    lexer.nextToken();
    expect(TokenType.LPar);
    Toke
    expe
    retu
  } else
    lexe
    expe
    Pars
    expe
    Pars
    expect(TokenType.RPar);
    return new BranchNode(left, right);
  } else {
    throw new Syn
  }
}
```

look ahead

Complete Java implementation of lexical analysis and recursive descent parsing for binary trees ***available on course website***

```
Token expect(TokenType t) throws SyntaxError {
  Token next = lexer.nextToken();
  if (next.getType() != t) throw new SyntaxError();
  return next; }
```

Eliminating Ambiguity

Ambiguous Expression Grammar

```
expr ::= intLiteral | ident
       | expr + expr | expr / expr
```

foo + 42 / bar + arg

Each node in parse tree is given by one grammar alternative.

Show that the input above has two parse trees!

(1) Layer the grammar by priorities

expr ::= ident | expr - expr | expr ^ expr | (expr)

expr ::= term (- term)*
term ::= factor (^ factor)*
factor ::= id | (expr)

lower priority binds weaker, so it goes outside

(2) Building trees: right-associative "^"

RIGHT-associative operator – using recursion
(or loop and then reverse a list)

x ^ y ^ z ➔ x ^ (y ^ z)

Exp(Var("x"), Exp(Var("y"), Var("z")))

```
Expr term() {
  Expr e = factor();
  if (lexer.token == ExpToken) {
    lexer.next();
    return new Exp(e, term());
  } else
    return e;
}
```

(3) Building trees: left-associative "-"

LEFT-associative operator

x - y - z → (x - y) - z

Minus(Minus(Var("x"), Var("y")), Var("z"))

```
Expr expr() {
  Expr e = term();
  while (lexer.token == MinusToken) {
    lexer.next();
    e = new Minus(e, term());
  }
  return e;
}
```

(3) Building trees: left-associative "-"

LEFT-associative operator

x - y - z → (x - y) - z

Minus(Minus(Var("x"), Var("y")), Var("z"))

```
Expr e
  Expr
  whil
    le
    e =
  }
  return e;
}
```

Complete Java implementation of lexical analysis and recursive descent parsing for arithmetic expressions ***available on course website***

Grammars & Ambiguity: Summary

- If we can find a string for which there are ***two different parse trees***, then the grammar is ambiguous
- In general, it is difficult to say whether a grammar is ambiguous, however
- Deciding whether a grammar is ambiguous is an ***undecidable problem***
  - There is no algorithm which can decide whether a grammar is ambiguous
- How to make grammars unambiguous:
  - Ensure that there is always ***only one parse tree***
  - Construct the ***correct*** abstract syntax tree (associativity etc.)

Manual Construction of Parsers

- Typically one applies previous transformations to get a nice grammar
- Then, we write recursive descent parser as set of mutually recursive procedures that check if input is well formed
- Then, enhance such procedures to construct trees, paying attention to the associativity and priority of operators

Grammar vs Recursive Descent Parser

```
expr ::= term termList
termList ::= + term termList
           |  - term termList
           | ε
term ::= factor factorList
factorList ::= * factor factorList
             | / factor factorList
             | ε
factor ::= name | ( expr )
name ::= ident
```

```
def expr = { term; termList }
def termList =
  if (token == PLUS) {
    skip(PLUS); term; termList
  } else if (token == MINUS)
    skip(MINUS); term; termList
  }
def term = { factor; factorList }

...

def factor =
  if (token == IDENT) name
  else if (token == LPAR) {
    skip(LPAR); expr; skip(RPAR)
  } else
    error("expected ident or (")
```

Recursive Descent: Summary

- One of the most ***widely-used*** methods for parsing in compilers of real-world programming languages
  - GCC C/C++ compiler, Java reference compiler, Scala reference compiler, …
- ***Efficient*** (linear) in the size of the token sequence
- ***Straight-forward to implement manually*** based on the grammar
  - There are also ***parser generators*** that generate the source code of recursive descent parsers
- Close ***correspondence between grammar and code***
  - Common practice: quote grammar in code comments