Abstract Syntax Trees
An Introduction to Grammars and Parsing Techniques
Grammars and Languages

- Grammars and Languages are one of the most established areas of Computer Science.
- Why are Grammars and Parsing Techniques relevant?
  - A grammar is a formal method to describe a (textual) language
    - Programming languages: C, Java, C#, JavaScript
    - Domain-specific languages: BibTeX, Mathematica
    - Data formats: log files, protocol data
  - Parsing: Tests whether a text conforms to a grammar
    - Turns a correct text into a parse tree (abstract syntax tree)
How to define a grammar?

- Simplistic solution: finite set of acceptable sentences
  - Problem: what to do with infinite languages?
- Realistic solution: finite recipe that describes all acceptable sentences
- A grammar is a finite description of a possibly infinite set of acceptable sentences
Suppose we want to describe a language that contains the following legal sentences:

- Egg
- Egg and Bacon
- Egg, Bacon and Sausage
- Egg, Sausage, Egg and Bacon
- ...

How do we find a finite recipe for this?
The Egg, Bacon and Sausage Grammar

- Meal \(\rightarrow\) Egg
- Meal \(\rightarrow\) Bacon
- Meal \(\rightarrow\) Sausage
- Sentence \(\rightarrow\) Meal
- Sentence \(\rightarrow\) List End
- List \(\rightarrow\) Meal
- List \(\rightarrow\) List, Meal
- End \(\rightarrow\) and Meal

Terminals:
- Egg
- Bacon
- Sausage
- ,
- and

Non-terminals:
- Meal
- Sentence
- List
- End

Start Symbol: Sentence
Variations in notation

- Name → **Egg** | **Bacon** | **Sausage**
- `<Name>` ::= “Egg” | “Bacon” | “Sausage”
Languages and Automata

- Regular languages
  - The weakest formal languages widely used
- Context-free languages
- Context-sensitive languages
- Recursive enumerable languages
## Operations on sets of words

<table>
<thead>
<tr>
<th>operation</th>
<th>notation</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>union</td>
<td>L ∪ M</td>
<td>L ∪ M = { s \mid s \in L \lor s \in M }</td>
</tr>
<tr>
<td>concatenation</td>
<td>L M</td>
<td>L M = { st \mid s \in L \land t \in M }</td>
</tr>
</tbody>
</table>
| self concatenation      | L^i      | \begin{align*} L^0 &= \{ \varepsilon \} \\
                       & L^1 &= L \\
                       & L^n &= LL^{n-1} \end{align*}                                      |
| Kleene closure          | L^*      | L^* = \bigcup_{i=0}^{\infty} L^i                                        |
| Kleene star             | L^*      |                                                                              |
| positive closure        | L^+      | L^+ = L L^*                                                                 |
| option                  | [ L ]    | [ L ] = L | \varepsilon               |

s, t  word, sequence of symbols
L, M  set of words
\varepsilon  empty word
Regular Expressions (RE)

- RE over alphabet $\Sigma$ define a **language** (is a set of words)
  1. $\varepsilon$ is a RE denoting the language $\{ \varepsilon \}$
  2. for every $a \in \Sigma$, $a$ is a RE denoting the language $\{ a \}$
  3. if $r$ and $s$ are REs denoting $L(r)$ and $L(s)$
     a) $(r | s) = L(r) \cup L(s)$
     b) $rs = L(r)L(s)$
     c) $(r)^* = L(r)^*$
- we use precedence to avoid parentheses: closure > concatenation > alternation
  '$_'$ | letter alpha$^*$ = ('$_'$ | letter (alpha)$^*$)
Examples

- letter → a|b|c|...|z|A|B|C|...|Z
- digit → 0|1|2|3|4|5|6|7|8|9
- ident → letter ( letter | digit )*

- integer → [+|−] (0|(1|2|3|4|5|6|7|8|9) digit*)
- decimal → integer . digit*
- real → (integer | decimal) E (+|−) digit+

- in most programming languages tokens can be described by REs

- note: names for REs are allowed, but definitions should not be recursive!
RLs and Finite Automata

THEOREM

L is regular

iff

L is accepted by a Finite Deterministic Automaton
Example DFA

letter → a|b|c|...|z|A|B|C|...|Z
digit → 0|1|2|3|4|5|6|7|8|9
ident → letter ( letter | digit )*
Building a recognizer for RLs

Recognizers can easily be encoded by hand.

```java
public enum Token {
    NumberToken ("<number>"),
    IdentToken ("<ident>"),
    PlusToken ("+"),
    MinusToken ("-"),
    DivToken ("/"),
    TimesToken ("*");

    private final String tokenString;

    Token (String str) { tokenString = str; }

    @Override
    public String toString () {
        return tokenString;
    }
}
```
Building a recognizer for RLs

```java
private void scan () throws IOException, ParseError {
    if ( currentChar == EOF_CHAR ) {
        currentToken = Token.EOFToken;
    } else {
        boolean simpleToken = trySimpleTokens ( Token.PlusToken, Token.MinusToken, Token.TimesToken,
                                               Token.DivToken, Token.OpenToken, Token.CloseToken );
        if ( ! simpleToken ) {
            if ( Character.isDigit( currentChar ) ) {
                scanNumber ( );
            } else if ( Character.isLetter( currentChar ) ) {
                scanIdent ( );
            } else {
                throw new ParseError ( "Illegal input character" );
            }
        }
    }
}
```
Limitations of Regular Languages

- Intuition: A finite automaton that runs long enough must repeat states.
- Finite automaton can’t remember # of times it has visited a particular state.
- Finite automaton has finite memory:
  - Only enough to store in which state it is.
  - Cannot count, except up to a finite limit.
- E.g., \( \{ (i)^i \mid i \geq 0 \} \) is not regular.
Context Free Grammars (CFG)

- CFGs allow recursive definitions of rules.
- Example: language of balanced parentheses

\[
\text{BPs} \rightarrow (\text{BPs})\text{BPs} \mid \varepsilon
\]

- Parsing: Checking if a sentence (e.g., a program) obeys given grammar rules and deriving the corresponding (structure) information.
Parsing

- A CFG can be used to generate strings in its language
  - “Given the CFG, construct a string that is in the language”
- A CFG can also be used to recognize strings in its language
  - “Given a string, decide whether it is in the language”
  - And, if it is, construct a derivation tree (or AST)

Parsing generally refers to this last step, i.e., going from a sentence (in the language) to its derivation tree or for a (programming) language perhaps to an AST for that sentence.
A parse tree (syntax tree, derivation tree) is a (rooted) tree that represents the syntactic structure of a sentence according to some CFG.

A parses usually constructs a parse tree

\[
E \rightarrow E \ast E \mid E + E \mid \text{num}
\]
A Recursive-Descent Parser

- One parse method per non-terminal symbol
- A non-terminal symbol on the right-hand side of a rewrite rule leads to a call to the parse method for that non-terminal
- A terminal symbol on the right-hand side of a rewrite rule leads to “consuming” that token from the input token string
- | in the CFG leads to “if-else” in the parser
Tokenizer

public Token peek ()

public void next () throws ParseError

public String getAttribute () throws ParseError

- **peek** retrieves, but does not remove, the head of the list of tokens. If there are no more tokens, peek yields **EOFToken**

- **next** removes the head from the list of tokens

- **getAttribute** yields the attribute of the token returned by peek. If the token has no attribute, it will throw a **ParseError** exception.

E.g., the attribute of a Number is its value
Example 1:

```
public BPs parseBPs() throws ParseError {
    if ( tokens.peek() == Token.OpenToken ) {
        tokens.next();
        BPs left = parseBPs();
        if ( tokens.peek() == Token.CloseToken ) {
            tokens.next();
            BPs right = parseBPs();
            return new NonEmptyBPs(left, right);
        } else {
            throw new ParseError(") expected", tokens.peek());
        }
    } else {
        return new EmptyBps();
    }
}
```
Example 1:

BPs → (BPs)BPs | ε

public BPs parseBPs() throws ParseError {
    if (tokens.peek() == Token.OpenToken) {
        tokens.next();
        BPs left = parseBPs();
        if (tokens.peek() == Token.CloseToken) {
            tokens.next();
            BPs right = parseBPs();
            return new NonEmptyBPs(left, right);
        } else {
            throw new ParseError(") expected", tokens.peek());
        }
    } else {
        throw new ParseError("Expected (", tokens.peek());
    }
} else {
    return new EmptyBps();
}
Abstract Syntax Tree

- An abstract base-class for each non-terminal symbol
- Concrete derived classes for each alternative
- Keywords/terminals are usually not included, unless they have an attribute.
public abstract class BPs {}

public class NonEmptyBPs extends BPs {
    private BPs left, right;

    public NonEmptyBPs ( BPs left, BPs right ) {
        this.left = left; this.right = right;
    }

    @Override
    public String toString () { return "(" + left + ")" + right; }
}

public class EmptyBps extends BPs {
    @Override
    public String toString () { return ""; }
}

Example 2: 

\[
E \rightarrow E \ast E \mid E + E \mid \text{num}
\]

```java
public Exp parseExp() throws ParseError {
    Exp left_arg = parseExp();
    if (tokens.peek() == TimesToken) {
        tokens.next();
        Exp right_arg = parseExp();
        return new TimesExp(left_arg, right_arg);
    } else if (tokens.peek() == PlusToken) {
        tokens.next();
        Exp right_arg = parseExp();
        return new TimesExp(left_arg, right_arg);
    } else {
        return left_arg;
    }
}
```

infinite loop
Left-recursion

- Recursive descent parsers cannot handle left-recursion in the grammar parsed
  - the parser uses the same rule recursively without consuming any input: infinite recursion
- Formally, a grammar is left-recursive if
  \[ \exists \text{ non-terminal } A \text{ such that } A \Rightarrow^+ A \alpha \text{ for some string } \alpha \]
- a simple expression grammar can be left-recursive:
  \[ \text{Expr} \rightarrow \text{Expr} + \text{Expr} \]
Eliminating left-recursion

- to remove left-recursion, we can transform the grammar
- consider the grammar fragment:
  \[ F \rightarrow F \alpha \ |
  \beta \ |
  \beta \]
  where \( \alpha \) and \( \beta \) do not start with \( F \)
- we can rewrite this grammar as:
  \[ F \rightarrow \beta F' \]
  \[ F' \rightarrow \alpha F' \ |
  \epsilon \]
  where \( F' \) is a new non-terminal
- this new grammar fragment contains no left-recursion
  - it does contain right-recursion
Eliminating left-recursion (2)

\[ E \rightarrow E \ast E \mid E + E \mid \text{num} \]

Becomes

\[ E \rightarrow \text{num} \ F \mid \text{num} \ G \]
\[ F \rightarrow \ast \ E \ F \mid \varepsilon \]
\[ G \rightarrow + \ E \ G \mid \varepsilon \]
Two more problems

The expression $1 \ast 2 + 3$ has two derivations

$$E \rightarrow E \ast E \mid E + E \mid \text{num}$$

ambiguous
Ambiguous grammars

\[

e \rightarrow \text{num} \ F \ | \ \text{num} \ G \\
F \rightarrow \ast \ E \ F \ | \ \varepsilon \\
G \rightarrow + \ E \ G \ | \ \varepsilon \\
\]

corresponds to

\[
\ast \\
\downarrow \\
1 + \\
\downarrow \\
2 \ 3
\]
Ambiguous grammars (2)

- Change your grammar according to common precedence rules:
  - add rule for each precedence class.

\[
\begin{align*}
E & \rightarrow T \ [+ \ E] \\
T & \rightarrow F \ [* \ T] \\
F & \rightarrow \text{num}
\end{align*}
\]

- \(12 \ * \ 34 \ + \ 5 \ * \ 6\) has as derivation: \((12 \ * \ 34) \ + \ (5 \ * \ 6)\)
Building parse trees: associativity

- for operators like + and * we can put the parenthesis in any way we want
  - e.g., $1 + 2 + 3 = (1 + 2) + 3 = 1 + (2 + 3)$

- for other operators this does not hold e.g.
  - $(3 - 2) - 1 = 1 - 1 = 0$
  - $3 - (2 - 1) = 3 - 1 = 2$
  - $(3 - 2) - 1 \neq 3 - (2 - 1)$
  - default way of parsing: $3 - 2 - 1 = (3 - 2) - 1$
  - similar $8 / 2 / 2$ should have value 2 (instead of 8)
associativity: parse trees

- parse trees for 3 - 2 - 1:
- $3 - 2 - 1 = (3 - 2) - 1 = 0$

```
3 - -
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>1</td>
</tr>
</tbody>
</table>

value: 2
```

```
- - 1
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

value: 0
associativity

- The rule
  \[ E \rightarrow T \ [ - \ E ] \]
  treats operators as if they were right associative

- solution: use
  \[ E \rightarrow T \ ( - \ T)^* \]
  ■ as soon as we recognized \( E \rightarrow T_1 - T_2 \ ( - \ T)^* \)
  we replace this by \( E \rightarrow T_3 \ ( - \ T)^* \) where \( T_3 = T_1 - T_2 \)
Example 3:

\[ E \rightarrow T \ ( - \ T)^* \]

```java
public Exp parseExp() throws ParseError {
    Exp t = parseTerm();
    for ( BinOp op = parseLowPrioOp();
         op != null;
         op = parseLowPrioOp() ) {
        t = new BinOpExp(t, parseTerm(), op);
    }
    return t;
}

public BinOp parseLowPrioOp() throws ParseError {
    if ( tokens.peek() == MinusToken ) {
        tokens.next();
        return new MinOp();
    } else {
        return null;
    }
}
```
AST traversal

- Our grammar

\[
\begin{align*}
E & \rightarrow T \ (L \ T)^* \\
T & \rightarrow F \ (H \ F)^* \\
F & \rightarrow \text{num} \\
& \quad | \ -\text{num} \\
& \quad | \ ( \ E \ ) \\
L & \rightarrow + \ | \ - \\
H & \rightarrow * \ | \ / \\
\end{align*}
\]

- Suppose we want to evaluate expressions.
Solution 1: dynamic binding

- Extend the base class with abstract method

```java
public abstract class Exp {
    public abstract int eval();
}

public class BinOpExp extends Exp {
    private Exp lo, ro;
    private BinOp op;

    public BinOpExp( Exp lo, Exp ro, BinOp op ) {
        this.lo = lo; this.ro = ro; this.op = op;
    }

    public int eval() {
        return op.eval( lo.eval(), ro.eval() );
    }
}
```

(eval is implemented in all (concrete) subclasses)
Solution 1

- For binary operations we introduce an interface
  ```java
  public interface BinOp {
    int eval(int lo, int ro);
  }
  ```

- With obvious implementations for all operators
  ```java
  public class PlusOp implements BinOp {
    @Override
    public int eval(int lo, int ro) {
      return lo + ro;
    }
  }
  ```

- solution relies on polymorphism
public class Eval {
    public int eval(Exp e) {
        if (e instanceof BinOpExp) {
            BinOpExp boe = (BinOpExp) e;
            BinOp bo = boe.getOp();
            if (bo instanceof PlusOp) {
                return eval(boe.getLeftArg()) + eval(boe.getRightArg());
            } else if (bo instanceof MinOp) {
                return eval(boe.getLeftArg()) - eval(boe.getRightArg());
            } else if (/* ... */){ /* ... */ }
        } else {
            NumExp ne = (NumExp) e;
            return ne.getValue();
        }
    }
}
Comparing both approaches

Tree traversals using abstract methods:

- **Pros:**
  - Easy to add a syntactic construct
  - Method dispatch is done automatically

- **Cons:**
  - Adding a new operation has a global effect
  - Syntax (representation) and semantics (operations) are interleaved
Comparing both approaches (2)

Tree traversals using `instanceof`

- **Pros:**
  - Easy to add a semantic function
  - Syntax and semantics are separated

- **Cons:**
  - Dispatch of semantic functions requires object casting
  - Adding a syntactic construct has a global effect
Visitor design pattern

Combine the best of both worlds:

- Easy to add a syntactic construct
- Easy to add a new methods
- Syntax and semantics are separated
- Automatic method dispatch
Visitor design pattern (2)

This is all provided by the visitor design pattern:

- Interface defines methods for each syntactic construct
- Semantic functions (new operations) are implementations of this interface
- Dispatch of operations using `call back requests`
The Visitor interface

Consider the AST class hierarchy for

```java
public abstract class Exp {
    public class PlusExp extends Exp {...}
    public class TimesExp extends Exp {...}
    public class NumExp extends Exp {...}
}
```

This results in the following interface

```java
public interface Visitor {
    public int visitPlus( PlusExp pe );
    public int visitTimes( TimesExp te );
    public int visitNum( NumExp ne );
}
```

One visit method for each concrete subclass
The Visitor interface

Consider the AST class hierarchy for

```java
public abstract class Exp { }
public class PlusExp extends Exp { ...}
public class TimesExp extends Exp { ...}
public class NumExp extends Exp { ...}
```

This results in the following interface

```java
public interface Visitor {
    public int visitPlus( PlusExp pe );
    public int visitTimes( TimesExp te );
    public int visitNum( NumExp ne );
}
```

Alternatively, we can use the same name for all methods

```java
public interface Visitor {
    public int visit( PlusExp pe );
    public int visit( TimesExp te );
    public int visit( NumExp ne );
}
```
Call back requests

Add an `accept` method to the class `Exp` (and its subclasses):

```java
public abstract class Exp {
    public int accept( Visitor v );
}

public class PlusExp extends Exp {
    ... 
    public int accept( Visitor v ) {
        return v.visit( this );
    }
}
```

Idea: you can ask an `Exp` to call back by invoking the `accept` method.
Implementing the Visitor interface

Consider an evaluation function of expressions:

```java
public class Evaluator implements Visitor {
    public int visit(PlusExp pe) {
        return pe.getLeftArg().accept(this) + pe.getRightArg().accept(this);
    }

    public int visit(TimesExp te) {
        return pe.getLeftArg().accept(this) * pe.getRightArg().accept(this);
    }

    public int visit(NumExp ne) {
        return ne.n;
    }
}
```
Adding different operations

Semantics functions (operations on the AST) often have different signatures.

- Evaluating expressions
  
  ```java
  public int visit( PlusExp pe ) ;
  ```

- Pretty-printing expressions
  
  ```java
  public String visit( PlusExp pe ) ;
  ```

- Optimizing expressions
  
  ```java
  public Exp visit( PlusExp pe ) ;
  ```

- Evaluating expressions with declarations
  
  ```java
  public int visit( PlusExp pe, Env e ) ;
  ```
Adding different operations (2)

Parameterize the Visitor interface using Java generics (or C++ templates)

```java
public interface GVisitor<R, A> {
    public R visit(PlusExp, A arg);
    public R visit(TimesExp, A arg);
    public R visit(NumExp, A arg);
}
```

```java
public abstract class Exp {
    public abstract <R, A> R accept(GVisitor<R, A> v, A arg);
}
```

```java
public class PlusExp extends Exp {
    ...
    public <R,A> R accept(GVisitor<R,A> v, A arg) {
        return v.visit(this, arg);
    }
}
```
A concrete visitor

This results in the following implementation for evaluation

```java
public class Evaluator implements GVisitor<Integer, Object> {
    @Override
    public Integer visit(PlusExp pe, Object arg) {
        return pe.getLeftArg().accept(this, arg) +
               pe.getRightArg().accept(this, arg);
    }

    ... 
    @Override
    public Integer visit(NumExp ne, Object arg) {
        return ne.getValue();
    }
}
```
String input_text = "1 + -7 - (4 + 3) * 5";

StringReader input = new StringReader( input_text );

RDparser parser = new RDparser(input);

Exp e = parser.parse();

Evaluator eval = new Evaluator();

System.out.printf( "Result: %d\n", e.accept( eval, null ) );
Next week

- Integer arithmetic