Compiling and Interpreting

- Interpreting a language consists of (at least) three stages
  - Lexical analysis
    - Reading individual characters and creating tokens: number, variable, reserved word, LEFT_PAREN, SEMI_COLO; ...
  - Parsing
    - Combining tokens into some structure representing a construct in the language, e.g., an abstract syntax tree (AST)
  - Executing/Interpreting
    - Use the AST, execute/interpret the construct represented
    - Differences between execution and interpretation?

- We can parse using well-known tools and techniques
  - Recursive-descent/built-by hand (top-down)
  - Top-down/Bottom-up parser generators: lex/yacc, javacc

Parsing and java.io.StreamTokenizer

- The class StreamTokenizer is designed to help with both lexical analysis and parsing
  - Recognizes numbers, words/identifiers, other symbols
  - Configurable as to case-sensitive, what’s a comment, is end-of-line significant, and so on
  - Not really object-oriented, but parsing might be simpler using a combination of OO and non-OO techniques
    - Leaves information in data field regarding the type of token parsed, e.g., number, word, EOL, EOF, char/symbol
    - Clients can extract number or word as appropriate

- The elan.parser examples use a StreamTokenizer with what’s essentially a one-token look-ahead
  - getToken() access, nextToken() advances and accesses

Recursive Descent for expressions

- We’ll use BNF (Backus-Naur form) for representing a valid expression

```
Expression ::= Term
   ::= Term +|- Expression
Term ::= Factor
   ::= Factor *|/ Term
Factor ::= Number
   ::= ( Expression )
   ::= - Factor
```

- This grammar represents expressions that conform to standard precedence rules. It’s a right-recursive grammar hence useful in top-down parsing
  - The BNF almost completely determines parsing rules, see elan.ExpressionParser.java

Parsing with tokens

- Each of the functions in ExpressionParser conforms:
  - Precondition: un-processed token accessible
  - Postcondition: token consumed, new token available

- Matching when parsing
  - Often we expect certain Tokens/symbols, use match to ensure parsing works
    - Example, in factor what happens after LEFT_PAREN?
    - Example, what happens in parsing a repeat expression?

- Each BNF rule usually corresponds to a parsing function, elements of grammar correspond to terminal/non-terminal nodes
  - Example: instructions, expressions
Use cases

See http://members.aol.com/acockburn/papers/AltIntro.htm

- An actor interacts with system
  - Might be a person using the system
  - Might be a program or process interacting with the system
- Use cases are descriptions of what happens when an actor uses the system to achieve a goal
  - Collection of possible sequences of interactions between the system and its actors relating to a goal
  - Use cases should define all system behavior relevant to actors achieving goals and should not involve other factors
- Easy to read, scenario/descriptive
- NOT: UI/GUI, implementation based

SLOGO use case

- There is a read-eval-print loop
  - How else might user interact with environment
  - What information should be accessible to user
  - What about display? Part of use case?
- Debugging and error messages, what are issues?
  - AST -> maps to source, what is this useful for?
  - Flag more than the first error? Error correction?
    - Throw exceptions?
- What about design/development/testing
  - How can graphical display be tested independently of parser, lexer?