Scanning and Parsing in Squeak

- Defining Lexical and Syntactic Analysis
  - Scanning/Tokenizing
  - Parsing
- Easy ways to do it
  - State Transition Tables
  - Recursive Descent Parsing
- More sophisticated way
  - T-Gen: Lex and YACC for Squeak
- Examples from Squeak
  - Smalltalk parser
  - HTML parser

Challenge of Compiling

- How do you go from source code to object code?
  - Lexical analysis: Figure out the pieces (tokens) of the program: Constants, variables, keywords.
  - Syntactic analysis: Figure out the meaning from structure (parsing)—and check structure
  - Backend: Generate object code

Lexical Analysis

- Given a bunch of characters, how do you recognize the key *things* and their types?
- Simplest way: Parse by spaces

```
'This is
a test
with returns in it.' findTokens: (Character
  cr asString),(Character space asString).

OrderedCollection ('This' 'is' 'a' 'test'
  'with' 'returns' 'in' 'it.' )
```

Scanning: Doing It Right

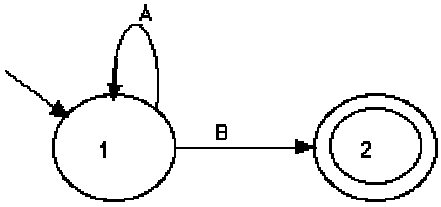
- Read in characters one-at-a-time
- Recognize when an important *token* has arrived
- Return the *type* and the *value* of the token

A Theoretical Tool for Scanning: FSA's

- Finite State Automata (FSA)
  - One model of computation that can scan well
  - We can make them fast and efficient
- FSA's are
  - A collection of states
  - Arcs between states
    - Labeled with input

Example FSA

- State 1 is start state
  - Incoming arrow
  - "Incomplete state" — can't end there
- State 2 is terminal state—can end there
- Consume A's in 1, end with a B in 2
  - Valid: AB, AAB, AAAB



General FSA Processing

- Enter the start state
- Read input
- Go to the state for that input
- If an End state, can end
  - But may not want to: Conflicts

Implementing FSAs

- Easiest way: State Transition Tables
  - Read a character
  - Append character to VALUE
  - Find state to move to given current STATE and input CHARACTER
  - If end state, return VALUE and STATE
    - (Sometimes need to lookahead. Could I grab the next character and be in another end state?)

Syntactic Analysis

- Given the tokens, can we recognize the *language*?
  - Parsing
- Structure for describing relationship between tokens is called a *grammar*
  - A grammar describes how tokens can be assembled into an acceptable sentence in a language
  - We're going to study a kind called *context-free grammars*

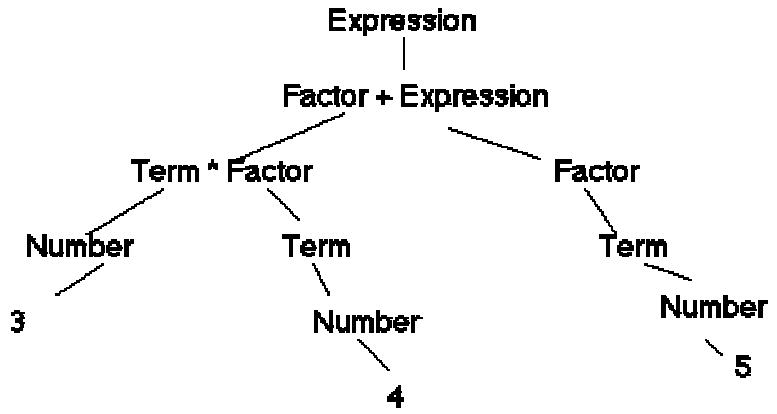
Context-free grammars

- Made up of a set of rules
- Each rule consists of a left-hand side *non-terminal* which maps to a right-hand side expression
- Expressions are made up of other *non-terminals and terminals*
- Rules can be used as replacements
  - Either side can be replaced with the other

Example grammar

- Expression := Factor + Expression
- Expression := Factor
- Factor := Term * Factor
- Factor := Term
- Term := Number
- Term := Identifier (*variable*)

Derivation tree using grammar for 3*4+5



Implementing Parsing

- Simplest way: Recursive descent parsing
  - Each non-terminal maps to a method/function/procedure in language
  - The m/f/p is responsible for recognizing the related non-terminal
    - Including calling one another as needed
  - Use your scanner to create tokens

A Simple Equation Recursive Descent Parser

- **Expression := Factor + Expression**
- **Expression := Factor**

expression

```
    Transcript show: 'Expression'; cr.
    self factor.
    (self peek = '+')
        ifTrue: [Transcript show: '+'; cr.
                 self pop.
                 self expression].
```

Factor and Term: Simple RD Parsing

- Factor := Term * Factor
- **Factor := Term**
- **Term := Number**

```
factor
    Transcript show: 'Factor'; cr.
    self term.
    (self peek = '*') ifTrue:
        [Transcript show: '*' ; cr.
        self pop.
        self factor.]
term
    Transcript show: 'Term' ; cr.
    (self nextIsNumber) ifTrue:
        [Transcript show: 'Number: ',(tokens first); cr.
        self pop.].
```

Simulating a Scanner

```
tokens: aCollection
    tokens := aCollection
peek
    ^tokens isEmpty
    ifTrue: [nil]
    ifFalse: [tokens first]
pop
    tokens := tokens allButFirst.
nextIsNumber
    ^(tokens first select: [:character |
        character asciiValue < $0 asciiValue or:
        [character asciiValue > $9 asciiValue]]) isEmpty
```

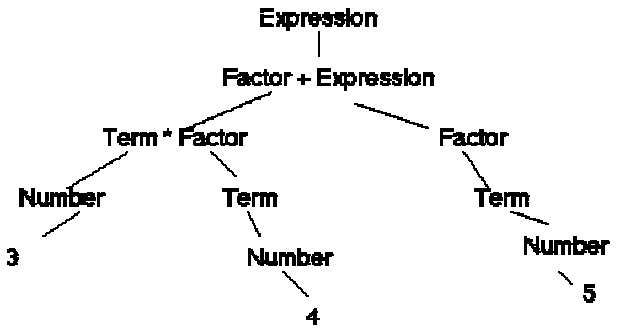
Trying out the toy parser

- eqn := EquationParser new.
- eqn tokens: ('3 * 4 + 5' findTokens: (Character space asString)).
- eqn expression.

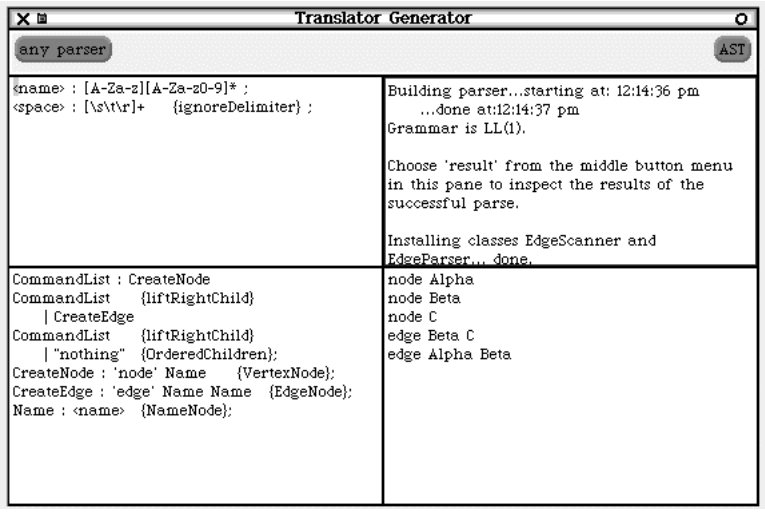
Comparing to the earlier derivation tree

- ***Transcript***:

Expression
Factor
Term
Number: 3
*
Factor
Term
Number: 4
+
Expression
Factor
Term
Number: 5



T-Gen: A Translator Generator for Squeak



Using T-Gen

- File in the changeSet
- In Morphic, **TGenUI open**
  - Enter your tokens as regular expressions in upper-left
  - Enter your grammar in lower-left
  - Put in sample code in lower-right
  - Transcript for parsing is upper-right
  - Processing of each occurs as soon as you accept (Alt/Cmd-S)
  - From the transcript pane, you can inspect *result*
  - Buttons let you specify *kind* of parser and *kind* of result
- You can *install* the resultant scanner and parser into your system

Walking the Graph Language Example

- (From user's manual in Zip: Kind of obtuse, so we'll walk it slowly here.)
- Entering the scanner:
  - <name> : [A-Za-z][A-Za-z0-9]* ;
  - <whitespace> : [\s\t\r]+ {ignoreDelimeter} ;
- ignoreDelimeter tells the system to drop these tokens
  - Tab before "{" is absolutely critical!

Creating our First T-Gen Grammar

- Smalltalk syntax for comments

```
CommandList : CreateNode
  CommandList
| CreateEdge CommandList
| "nothing" ;
CreateNode : 'node' <name> ;
CreateEdge : 'edge' <name> <name> ;
```

Toy input

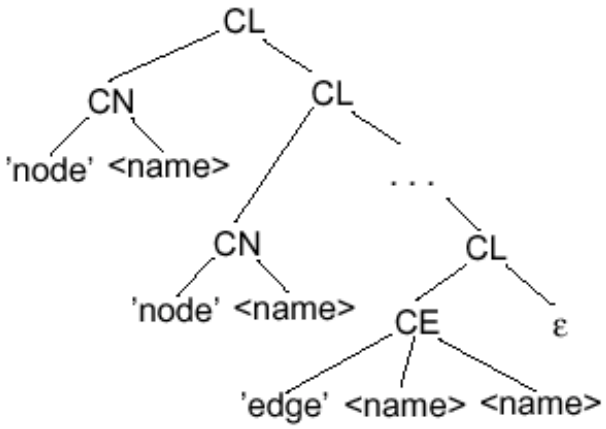
- node Alpha
- node Beta
- node C
- edge Beta C
- edge Alpha Beta

Generating Derivation Tree

- Only option with a limited grammar

```
CommandList
. CreateNode
. . 'node'
. . '<name>'
. CommandList
. . CreateNode
. . . 'node'
. . . '<name>'
. . CommandList
. . . CreateNode
. . . . 'node'
. . . . '<name>'
. . . CommandList
. . . . CreateEdge
. . . . . 'edge'
. . . . . '<name>'
. . . . . '<name>'
. . . . CommandList
. . . . . CreateEdge
. . . . . . 'edge'
. . . . . . '<name>'
. . . . . . '<name>'
. . . . . CommandList
. . . . . . '<epsilon>'
```

Generating Derivation Tree



But that's not too helpful

- That tells us the components, but not the values
- To get further, we want an Abstract Syntax Tree, or even to call our own methods on each grammar rule
- But we'll need to restructure the grammar so that each rule produces something useful.

Re-Writing the Grammar to generate an AST

```
CommandList : CreateNode CommandList
   {liftRightChild}
| CreateEdge CommandList {liftRightChild}
| "nothing" {OrderedChildren} ;
CreateNode : 'node' Name {VertexNode} ;
CreateEdge : 'edge' Name Name {EdgeNode}
   ;
Name : <name> {NameNode} ;
```

What's Going On Here?

- liftRightChild means "don't do anything here yet—just recognize the form"
- OrderedChildren is a pre-defined class that simply gathers everything (here, "nothing") into an OrderedCollection
- The rest are all classes that WE have to build

Providing the Appropriate Classes for AST

- Our classes must inherit from **ParseTreeNode**
  - Our classes are going to represent the nodes of the parse tree
- We must override methods to record the terminals passed in
  - **setAttribute: aString** to capture single terminals in a grammar rule
  - **addChildrenInitial: anOrderedCollection** to capture multiple terminals

NameNode

```
ParseTreeNode subclass: #NameNode
   instanceVariableNames: 'name '
   classVariableNames: ''
   poolDictionaries: ''
   category: 'edge-parser'
setAttribute: aString
   name := aString.
   Transcript show: 'NameNode: ',aString; cr.
```

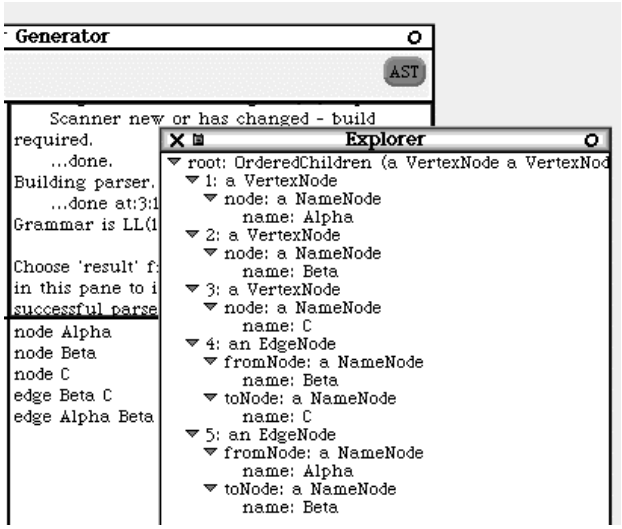
VertexNode

```
ParseTreeNode subclass: #VertexNode
    instanceVariableNames: 'node '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'edge-parser'
addChildrenInitial: anOrderedCollection
    anOrderedCollection size = 1
        ifTrue: [node := anOrderedCollection first.
                Transcript show: 'VertexNode: ',node
    printString; cr]
        ifFalse: [self error: 'VertexNode: Wrong number of
    children']
```

EdgeNode

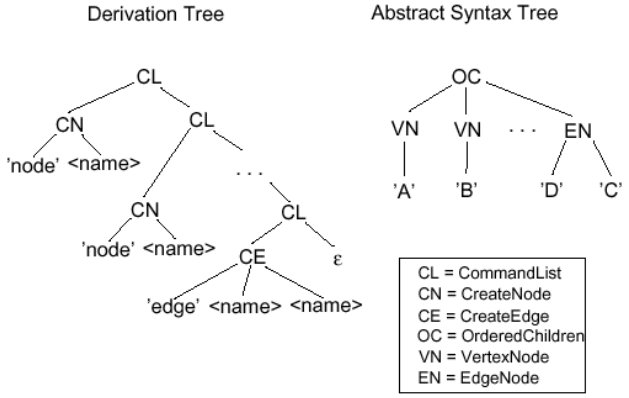
```
ParseTreeNode subclass: #EdgeNode
    instanceVariableNames: 'fromNode toNode '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'edge-parser'
addChildrenInitial: anOrderedCollection
    anOrderedCollection size = 2
    ifTrue:
        [fromNode := anOrderedCollection removeFirst.
        toNode := anOrderedCollection first.
        Transcript show: 'EdgeNode: ',fromNode printString,'--
    >',toNode printString; cr.]
    ifFalse: [self error: 'EdgeNode: Wrong number of children']
```

Now we can generate AST



AST: Abstract Syntax Tree

- Is useful to have a single data structure with all the components usefully identified



But AST's Are All-At-Once

- Alternatively, can capture elements as-they-are-parsed
- Transcript from AST generation:

```
VertexNodeNameNode: Beta
NameNode: Alpha
EdgeNode: a NameNode-->a NameNode
NameNode: C
NameNode: Beta
EdgeNode: a NameNode-->a NameNode
NameNode: C
VertexNode: a NameNode
NameNode: Beta
VertexNode: a NameNode
NameNode: Alpha
VertexNode: a NameNode
```

```
node Alpha
node Beta
node C
edge Beta C
edge Alpha Beta
```

T-Gen will call methods for you

```
CommandList : CommandList CreateNode
  {toGraph:addVertex:}
| CommandList CreateEdge
  {toGraph:addEdge:}
| "nothing" {createGraph} ;
CreateNode : 'node' Name
  {createVertexLabeled:} ;
CreateEdge : 'edge' Name Name
  {edgeFrom:to:} ;
Name : <name> {answerArgument:} ;
```

Note Careful Construction of Grammar and Methods

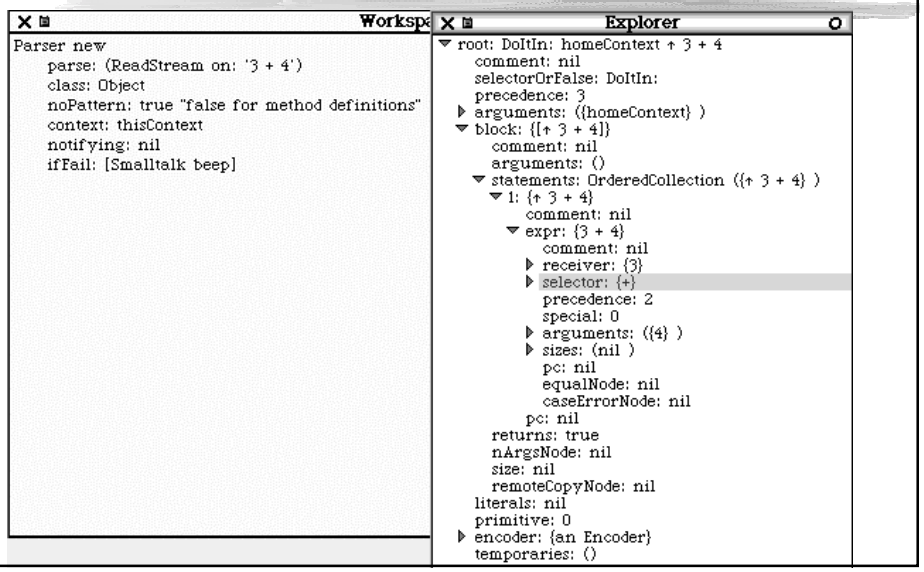
- Notice in the AST Construction
  - Things are assembled in reverse!
  - Grammar is recognized, and then built bottom-up
- Thus, you build things at the terminals, and assemble them into structures further up the tree
  - Thus, createGraph at the bottom
  - toGraph:addXXXX: in the mid-level rules

Once parser is built, install it

- You provide a name, e.g., Edge, and EdgeParser and EdgeScanner are generated
- Generating an AST:

```
(EdgeParser new parseForAST: '
node Alpha
node Beta
node C
edge Beta C
edge Alpha Beta
' ifFail: [Smalltalk beep.])
```

Smalltalk Parser



Smalltalk's Parser is Recursive Descent!

- Scanner methods are in Parser
  - Scanning method category: advance endOfLastToken match: matchToken: startOfNextToken
- All the kinds of messages are defined in Expression Types
  - argumentName assignment: blockExpression braceExpression cascade expression messagePart:repeat: method:context: pattern:inContext: primaryExpression statements:innerBlock: temporaries temporaryBlockVariables variable

Example: Parsing an Assignment

```
assignment: varNode
    " var ':=' expression => AssignmentNode."
    | loc |
    (loc := varNode assignmentCheck: encoder at: prevMark +
    requestorOffset) >= 0
        ifTrue: [^self notify: 'Cannot store into' at: loc].
    varNode nowHasDef.
    self advance.
    self expression ifFalse: [^self expected: 'Expression'].
    parseNode := AssignmentNode new
                        variable: varNode
                        value: parseNode
                        from: encoder.
    ^true
```

AssignmentNode then generates the code

- emitForValue:on: generates the bytecodes for the assignment

```
emitForValue: stack on: aStream
    value emitForValue: stack on: aStream.
    variable emitStore: stack on: aStream
```

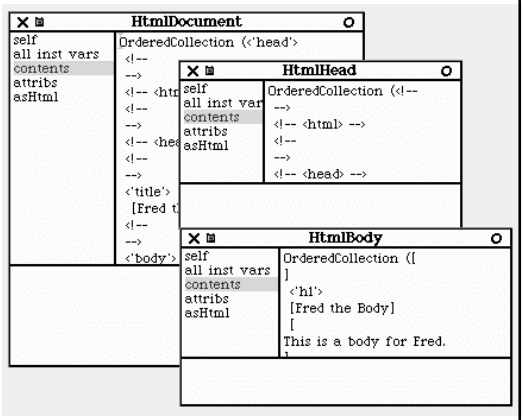
HtmlParser

- Used for Scamper

```
HtmlParser parse: '
<html>
<head>
<title>Fred the Page</title>
</head>
<body>
<h1>Fred the Body</h1>
This is a body for Fred.
</body>
</html>'
```

HtmlParser returns an HtmlDocument

- HtmlDocument has contents, which is an OrderedCollection
  - HtmlHead
  - HtmlBody
- HtmlEntity Hierarchy exists



Walk the Object Structure

```
doc := HtmlParser parse: '
<html>
<head>
<title>Fred the Page</title>
</head>
<body>
<h1>Fred the Body</h1>
This is a body for Fred.
</body>
</html>'.
```

```
body := doc contents last.
    "This should be an
    HtmlBody"
body contents detect:
    [:entity | entity isKindOf:
    HtmlHeader]. "This
    should be the first
    heading."
```

PrintIt:

```
<'h1'>
 [Fred the Body]
```

Summary

- Two key activities: Lexical analysis (tokenizing) and syntactic analysis (parsing)
  - Tokenizing techniques: FSAs and State Transition Tables
  - Parsing techniques
    - Recursive Descent Parsing
    - Table-driven Parsing (e.g., YACC, T-Gen)
- Example parsers
  - Simple equation parser
    - tinyHTML in examples file
  - T-Gen Graph language parser
  - SmalItalk's parser
  - HtmlParser in Squeak for Scamper