Parsing with Pushdown Automata

Theory of Computation

CISC 603, Spring 2020, Daqing Yun

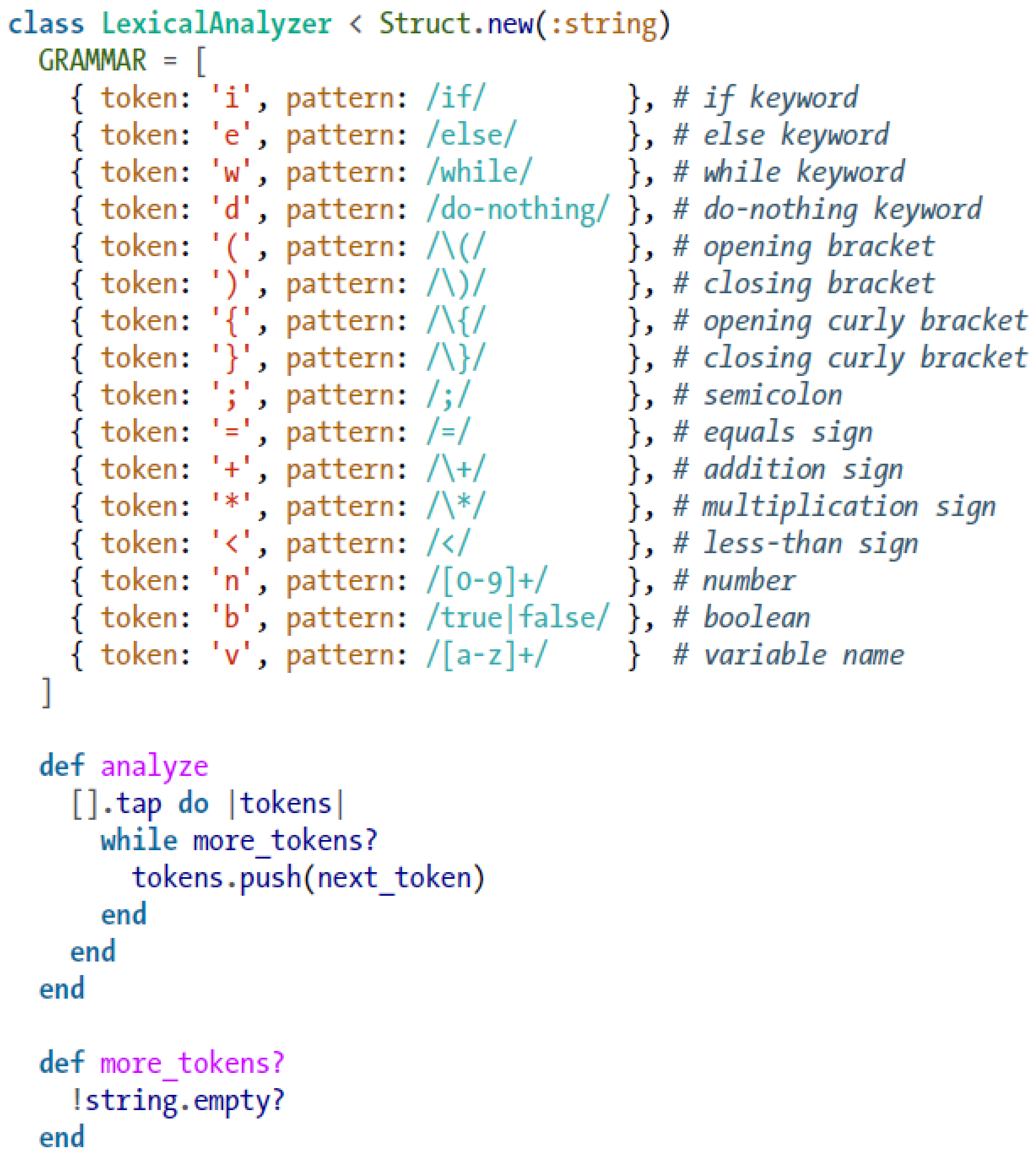
Recall – Lexical Analysis

- Read a raw string of characters and turn it into a sequence of *tokens* – each token represents an individual building block of the program syntax, like variable name, opening bracket, while keyword, etc.
- A lexical analyzer uses a language-specific set of rules called a *lexical grammar* to decide which character sequences should produce which tokens
- This stage deal with messy character-level details like variable-naming rules, comments, white spaces, leaving a clean sequence of tokens for the next stage to consume

Recall – Lexical Analysis

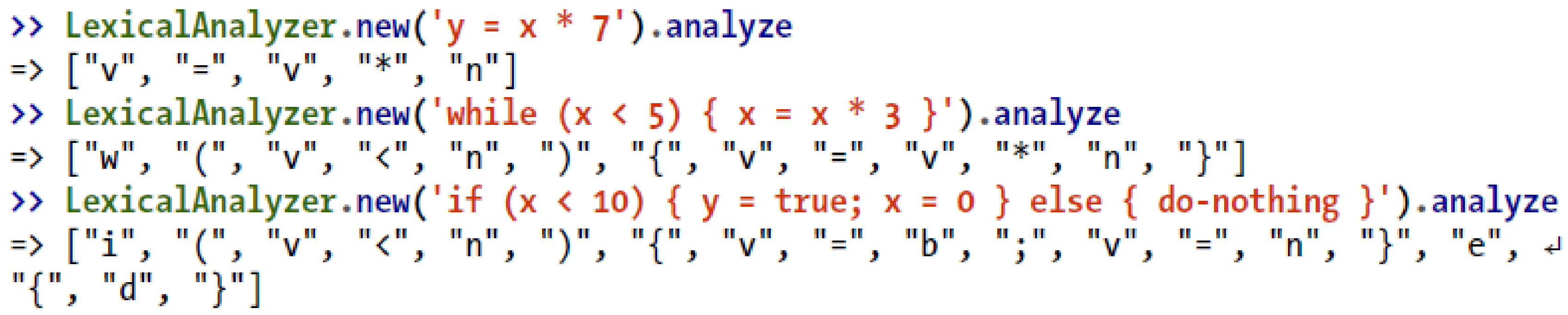
- The lexical analysis stage is usually straightforward and can be done with regular expressions (and therefore by NFAs)
- It involves simply matching a flat sequence of characters against some *rules* and deciding whether those characters look like a keyword, a variable name, an operator, or whatever else

Chop up a simple program into tokens



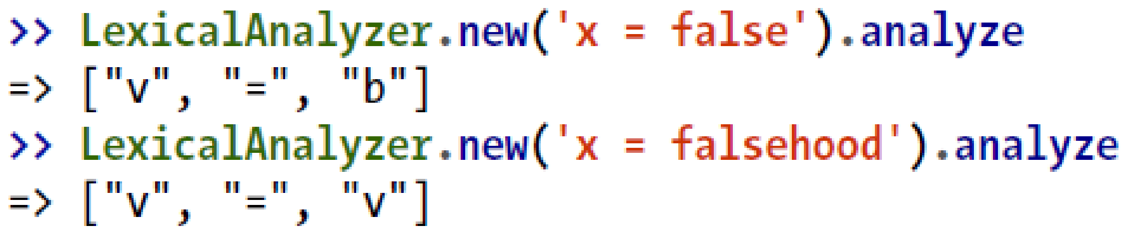
```ruby
class LexicalAnalyzer < Struct.new(:string)
  GRAMMAR = [
    { token: 'i', pattern: /if/         }, # if keyword
    { token: 'e', pattern: /else/       }, # else keyword
    { token: 'w', pattern: /while/      }, # while keyword
    { token: 'd', pattern: /do-nothing/ }, # do-nothing keyword
    { token: '(', pattern: /\(/         }, # opening bracket
    { token: ')', pattern: /\)/         }, # closing bracket
    { token: '{', pattern: /\{/         }, # opening curly bracket
    { token: '}', pattern: /\}/         }, # closing curly bracket
    { token: ';', pattern: /;/          }, # semicolon
    { token: '=', pattern: /=/          }, # equals sign
    { token: '+', pattern: /\+/         }, # addition sign
    { token: '*', pattern: /\*/         }, # multiplication sign
    { token: '<', pattern: /</          }, # less-than sign
    { token: 'n', pattern: /[0-9]+/     }, # number
    { token: 'b', pattern: /true|false/ }, # boolean
    { token: 'v', pattern: /[a-z]+/     }  # variable name
  ]

  def analyze
    [].tap do |tokens|
      while more_tokens?
        tokens.push(next_token)
      end
    end
  end

  def more_tokens?
    !string.empty?
  end
  def next_token
    rule, match = rule_matching(string)
    self.string = string_after(match)
    rule[:token]
  end

  def rule_matching(string)
    matches = GRAMMAR.map { |rule| match_at_beginning(rule[:pattern], string) }
    rules_with_matches = GRAMMAR.zip(matches).reject { |rule, match| match.nil? }
    rule_with_longest_match(rules_with_matches)
  end

  def match_at_beginning(pattern, string)
    /\A#{pattern}/.match(string)
  end

  def rule_with_longest_match(rules_with_matches)
    rules_with_matches.max_by { |rule, match| match.to_s.length }
  end

  def string_after(match)
    match.post_match.lstrip
  end
end
```

By creating a *LexicalAnalyzer* instance with a string of simple code and calling its *analyze* method, we can get back an array of tokens showing how the code breaks down into keywords, operators, punctuation, and other pieces of syntax:



```
>> LexicalAnalyzer.new('y = x * 7').analyze
=> ["v", "=", "v", "*", "n"]
>> LexicalAnalyzer.new('while (x < 5) { x = x * 3 }').analyze
=> ["w", "(", "v", "<", "n", ")", "{", "v", "=", "v", "*", "n", "}"]
>> LexicalAnalyzer.new('if (x < 10) { y = true; x = 0 } else { do-nothing }').analyze
=> ["i", "(", "v", "<", "n", ")", "{", "v", "=", "b", ";", "v", "=", "n", "}", "e", ↵
"{", "d", "}"]
```

- Choosing the rule with the *longest* match allows the lexical analyzer to handle variables whose names would otherwise cause them to be wrongly identified as keywords:

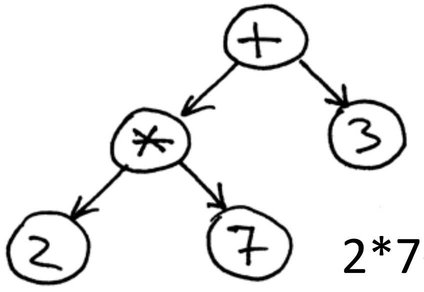


```
>> LexicalAnalyzer.new('x = false').analyze
=> ["v", "=", "b"]
>> LexicalAnalyzer.new('x = falsehood').analyze
=> ["v", "=", "v"]
```

- There are other ways of dealing with this problem, e.g., write more restrictive regular expressions in the rules: if the Boolean rule used the pattern /(true|false)(?![a-z])/, then it would not match the string *falsehood* in the first place

Recall – Syntactic Analysis

- Read a sequence of tokens and decide whether they represent a valid program according to the *syntactic grammar* of the language being parsed
- If the program is valid, the syntactic analyzer may produce additional information about its structure (e.g., a parse tree)



2*7+3

Recall – Syntactic Analysis

- How to decide whether those tokens represent a syntactically valid program keywords
- Cannot use regular expressions of NFAs because our program’s syntax is designed to allow *arbitrary nesting* of brackets
- Finite Automata are not powerful enough to recognize languages like that
- The use of PDA comes in handy to recognize valid sequences of tokens

- We need a syntactic grammar that describes how tokens may be combined to form programs

a SIMPLE statement is either a while loop or an assignment

```
<statement>  ::= <while> | <assign>
<while>      ::= 'w' '(' <expression> ')' '{' <statement> '}'
<assign>     ::= 'v' '=' <expression>
<expression> ::= <less-than>
<less-than>  ::= <multiply> '<' <less-than> | <multiply>
<multiply>   ::= <term> '*' <multiply> | <term>
<term>       ::= 'n' | 'v'
```

an assignment statement consists of a variable name followed by an equal sign and an expression

- Context-Free Grammar (CFG)
- Each rule has a *symbol* on the left-hand side and one or more sequences of symbols and tokens on the right – derivation rule

```
<statement>  ::= <while> | <assign>
<while>      ::= 'w' '(' <expression> ')' '{' <statement> '}'
<assign>     ::= 'v' '=' <expression>
<expression> ::= <less-than>
<less-than>  ::= <multiply> '<' <less-than> | <multiply>
<multiply>   ::= <term> '*' <multiply> | <term>
<term>       ::= 'n' | 'v'
```

- The CFG is a static description of a program’s structure, but we can also think of it as a set of rules for *generating* simple programs
- Starting from the <statement> symbol, we can apply the grammar rules to recursively expand symbols until only tokens remain

- Here’s one of many ways to fully expand <statement> according to the rules:

```
<statement> → <assign>
            → 'v' '=' <expression>
            → 'v' '=' <less-than>
            → 'v' '=' <multiply>
            → 'v' '=' <term> '*' <multiply>
            → 'v' '=' 'v' '*' <multiply>
            → 'v' '=' 'v' '*' <term>
            → 'v' '=' 'v' '*' 'n'
```

- This tells us that 'v' '=' 'v' '*' 'n' represents a syntactically valid program, but we want the ability to go in the opposite direction: to *recognize* valid programs, not generate them

- We need to figure out a way that can turn a context-free grammar into a nondeterministic pushdown automaton (NPDA)

- When we get a sequence of tokens out of the lexical analyzer, we’d like to know whether it’s possible to expand the <statement> symbol into those tokens by applying the grammar rules in some order

Converting CFG to PDA

1. Pick a character to represent each symbol from the grammar. In this case, we will use the uppercase initial of each symbol, S for <statement>, W for <while>, and so on, to distinguish them from the lowercase characters that we are using as tokens

Converting CFG to PDA

2. Use the PDA’s stack to store characters that represent grammar symbols (S, W, A, E, …) and tokens (w, v, =, *, …). When the PDA starts, let it immediately push a symbol onto the stack to represent the structure it is trying to recognize. We want to recognize program statements, so our PDA will begin by pushing S onto the stack:

```
>> start_rule = PDARule.new(1, nil, 2, '$', ['S', '$'])
=> #<struct PDARule …>
```

$\underbrace{['S', '\$']}_{\text{PDA stack}}$

Converting CFG to PDA

3. Translate the grammar rules into PDA rules that expand symbols on the top of the stack without reading any input
   - Each grammar rule describes how to expand a single symbol into a sequence of other symbols and tokens, and we can turn that description into a PDA rule that pops a particular symbol's character off the stack and pushes other characters on:

```
>> symbol_rules = [
     # <statement> ::= <while> | <assign>
     PDARule.new(2, nil, 2, 'S', ['W']),
     PDARule.new(2, nil, 2, 'S', ['A']),

     # <while> ::= 'w' '(' <expression> ')' '{' <statement> '}'
     PDARule.new(2, nil, 2, 'W', ['w', '(', 'E', ')', '{', 'S', '}']),

     # <assign> ::= 'v' '=' <expression>
     PDARule.new(2, nil, 2, 'A', ['v', '=', 'E']),

     # <expression> ::= <less-than>
     PDARule.new(2, nil, 2, 'E', ['L']),

     # <less-than> ::= <multiply> '<' <less-than> | <multiply>
     PDARule.new(2, nil, 2, 'L', ['M', '<', 'L']),
     PDARule.new(2, nil, 2, 'L', ['M']),

     # <multiply> ::= <term> '*' <multiply> | <term>
     PDARule.new(2, nil, 2, 'M', ['T', '*', 'M']),
     PDARule.new(2, nil, 2, 'M', ['T']),

     # <term> ::= 'n' | 'v'
     PDARule.new(2, nil, 2, 'T', ['n']),
     PDARule.new(2, nil, 2, 'T', ['v'])
   ]
=> [#<struct PDARule …>, #<struct PDARule …>, …]
```

Converting CFG to PDA

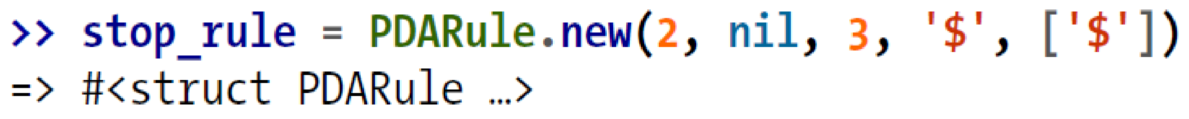
4. Give every token character a PDA rule that reads that character from the input and pops it off the stack:

```
>> token_rules = LexicalAnalyzer::GRAMMAR.map do |rule|
     PDARule.new(2, rule[:token], 2, rule[:token], [])
   end
=> [#<struct PDARule …>, #<struct PDARule …>, …]
```

These token rules work in opposition to the symbol rules. The symbol rules tend to make the stack larger, sometimes pushing several characters to replace the one that's been popped; the token rules always make the stack smaller, consuming input as they go

Converting CFG to PDA

5. Finally, make a PDA rule that will allow the machine to enter an accept state if the stack becomes empty:



```
>> stop_rule = PDARule.new(2, nil, 3, '$', ['$'])
=> #<struct PDARule …>
```

- Now we can build a PDA with these rules and feed it a string of tokens to see whether it recognizes them. The rules generated by the program grammar are nondeterministic as there are multiple applicable rules whenever the character S, L, M, or T is topmost on the stack, so it’ll have to be an NPDA:

```
>> rulebook = NPDARulebook.new([start_rule, stop_rule] + symbol_rules + token_rules)
=> #<struct NPDARulebook rules=[…]>
>> npda_design = NPDADesign.new(1, '$', [3], rulebook)
=> #<struct NPDADesign …>
>> token_string = LexicalAnalyzer.new('while (x < 5) { x = x * 3 }').analyze.join
=> "w(v<n){v=v*n}"
>> npda_design.accepts?(token_string)
=> true
>> npda_design.accepts?(LexicalAnalyzer.new('while (x < 5 x = x * }').analyze.join)
=> false
```

'w(v<n){v=v*n}'

State	Accepting?	Stack contents	Remaining input	Action
1	no	$	w(v<n){v=v*n}	push S, go to state 2
2	no	S$	w(v<n){v=v*n}	pop S, push W
2	no	W$	w(v<n){v=v*n}	pop W, push w(E){S}
2	no	w(E){S}$	w(v<n){v=v*n}	read w, pop w
2	no	(E){S}$	(v<n){v=v*n}	read (, pop (
2	no	E){S}$	v<n){v=v*n}	pop E, push L
2	no	L){S}$	v<n){v=v*n}	pop L, push M<L
2	no	M<L){S}$	v<n){v=v*n}	pop M, push T
2	no	T<L){S}$	v<n){v=v*n}	pop T, push v
2	no	v<L){S}$	v<n){v=v*n}	read v, pop v
2	no	<L){S}$	<n){v=v*n}	read <, pop <
2	no	L){S}$	n){v=v*n}	pop L, push M
2	no	M){S}$	n){v=v*n}	pop M, push T
2	no	T){S}$	n){v=v*n}	pop T, push n
2	no	n){S}$	n){v=v*n}	read n, pop n

'w(v<n){v=v*n}'

State	Accepting?	Stack contents	Remaining input	Action
2	no	){S}$	){v=v*n}	read), pop)
2	no	{S}$	{v=v*n}	read {, pop {
2	no	S}$	v=v*n}	pop S, push A
2	no	A}$	v=v*n}	pop A, push v=E
2	no	v=E}$	v=v*n}	read v, pop v
2	no	=E}$	=v*n}	read =, pop =
2	no	E}$	v*n}	pop E, push L
2	no	L}$	v*n}	pop L, push M
2	no	M}$	v*n}	pop M, push T*M
2	no	T*M}$	v*n}	pop T, push v
2	no	v*M}$	v*n}	read v, pop v
2	no	*M}$	*n}	read *, pop *
2	no	M}$	n}	pop M, push T
2	no	T}$	n}	pop T, push n
2	no	n}$	n}	read n, pop n

'w(v<n){v=v*n}'

State	Accepting?	Stack contents	Remaining input	Action
2	no	}$	}	read }, pop }
2	no	$		go to state 3
3	yes	$		—

- This execution trace shows us how the machine ping-pongs between symbol and token rules: the symbol rules repeatedly expand the symbol on the top of the stack until it gets replaced by a token, then the token rules consume the stack (and the input) until they hit a symbol
- This back and forth eventually results in an empty stack as long as the input string can be generated by the grammar rules

→ ***LL parsing***

Practicalities

- This parsing procedure relies on nondeterminism, but in real applications, it is better to avoid nondeterminism, because a deterministic PDA is much faster and easier to simulate
- Fortunately, it is almost always possible to eliminate nondeterminism by using the input tokens themselves to make decisions about which symbol rule to apply at each stage—a technique called *lookahead*—but that makes the translation from CFG to PDA more complicated

Practicalities

- It is also not good enough to only be able to *recognize* valid programs
- The whole point of parsing a program is to turn it into a structured representation that we can then do something useful with
- In practice, we can create this representation by instrumenting our PDA simulation to record the sequence of rules it follows to reach an accept state, which provides enough information to construct a *parse tree*

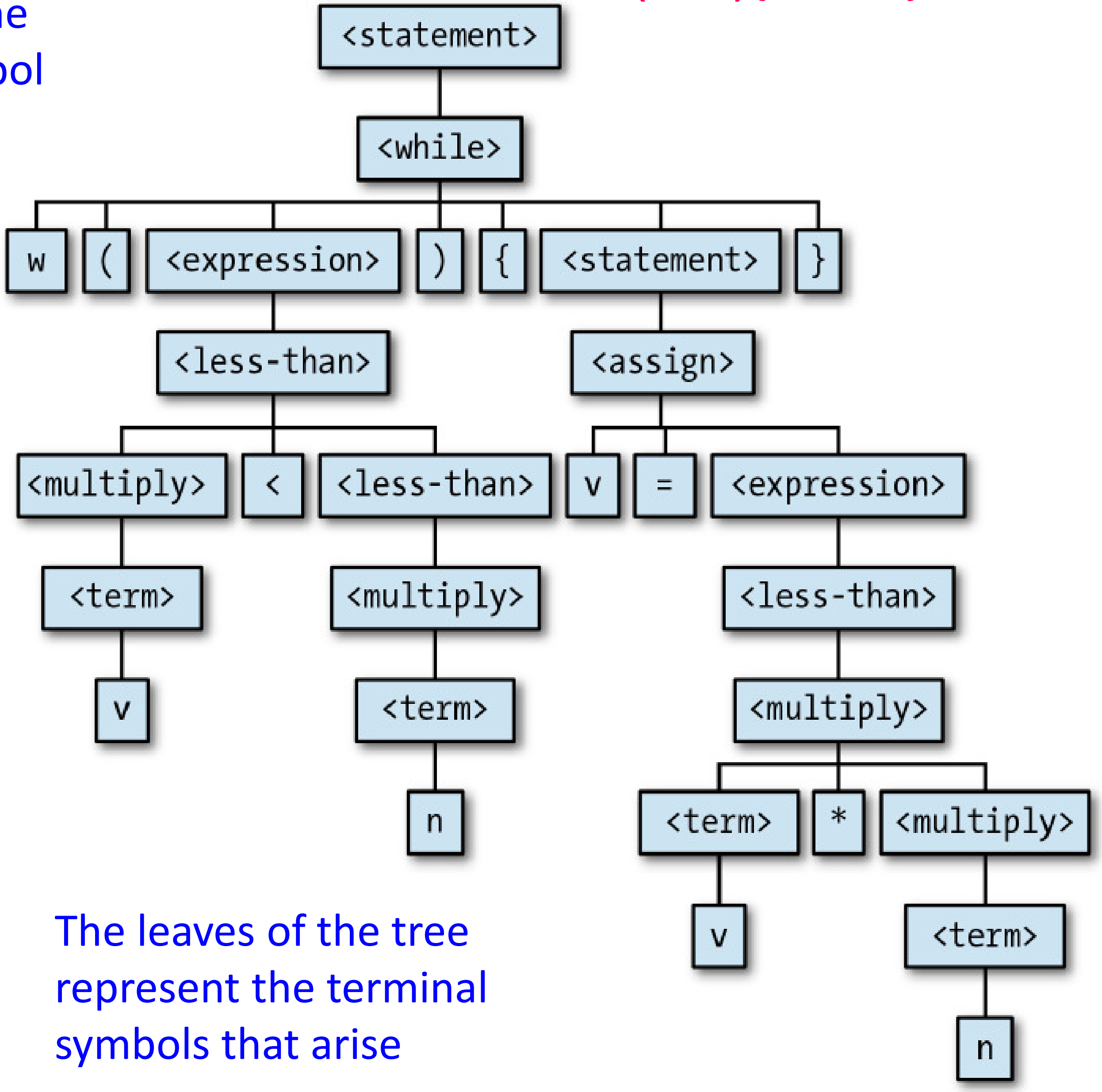
Practicalities

- For example, the previous execution trace shows us how the symbols on the stack get expanded to form the desired sequence of tokens, and that tells us the shape of the parse tree for the string 'w(v<n){v=v*n}':

'w(v<n){v=v*n}'

The root is the starting symbol

The internal vertices of the tree represent the non-terminal symbols that arise in the derivation



The leaves of the tree represent the terminal symbols that arise

How Much Power?

- DPDAs are more powerful than DFAs and NFAs; NPDAs are even more powerful
- Having access to a stack gives PDA more power and sophistication than FAs
- The main consequence of having a stack is the ability to recognize certain languages that finite automata aren’t capable of recognizing, like palindromes and strings of balanced brackets

How Much Power?

- The unlimited storage provided by a stack lets a PDA remember arbitrary amounts of information during a computation and refer back to it later
- A PDA can loop indefinitely without reading any input. A DFA can only ever change state by consuming a character of input
  - Although an NFA can change state spontaneously by following a free move, it can only do that a finite number of times before it ends up back where it started

How Much Power?

- Pushdown automata can also control themselves to a limited extent. There’s a *feedback loop* between the rules and the stack—*the contents of the stack affect which rules the machine can follow, and following a rule will affect the stack contents*—which allows a PDA to store away information on the stack that will influence its future execution

What are PDAs’ limitations?

- PDAs are seriously limited by the way a stack works, even if we are only interested in the kinds of pattern-matching applications as we have seen
- There’s no random access to stack contents below the topmost character
  - If a machine wants to read a character that’s buried halfway down the stack
  - It must pop everything above it – once characters have been popped, they’re gone forever

What are PDAs' limitations?

- PDAs can recognize palindromes, but they can't recognize doubled-up strings like '*abab*' and '*baaabaaa*', because once information has been pushed onto a stack, it can only be consumed in reverse order
- PDAs, NFAs, DFAs – what if we use them as a model of general-purpose computers

A Model of General-Purpose Computers?

- DFAs, NFAs, and PDAs are still a long way from being properly useful
- None of them has a decent *output* mechanism
  - They can communicate success by going into an accept state, but can't output even a single character (much less a whole string of characters) to indicate a more detailed result

A Model of General-Purpose Computers?

- This inability to send information back out into the world means that they can’t implement even a simple *algorithm* like adding two numbers together
- Like finite automata, an individual PDA has a fixed program
- There isn’t an obvious way to build a PDA that can somehow read a program from its input and *run it*

A Model of General-Purpose Computers?

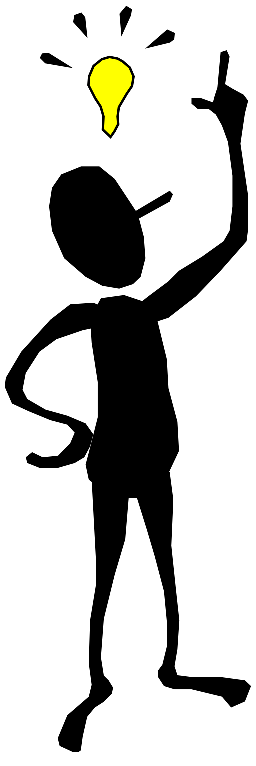
- All of these weaknesses mean that we need a better model of computation to really investigate what computers are capable of
- That is exactly what the next lecture is about – Turing Machines

Course Project –
A Toy Compiler

- Write a program (using any programming language) that can "understand" and "execute" some pre-defined commands

```
// this is a curve of sin function
origin is (200,200);
rot is 0;
scale is (10,4);
for T from 0 to 2*pi + pi/50 step pi/500 draw(T,-30*sin(T));
```

- Prepare to present, discuss, and take questions about your "compiler" in 10–15 minutes in class during our 3rd executive visit



Thanks ! ☺

Questions ?