Source Code

To humans: immediately conveys syntactic structure

To computers: a sequence of chars

Lexing: turning streams of chars into tokens

Tokens: grammatical symbols in computer-readable form

Parsers: discover the grammatical structure of (tokenized) programs

Regular expressions define regular languages, FA recognize

Grammars define context-free languages, parsers recognize

Programming languages syntax specified by grammars
Grammars and regular expressions:
Both define languages

Grammars are more powerful:
Describe langs not described by regexp

\[ exp := \text{id} | \text{exp} + \text{exp} | \text{exp} - \text{exp} | ( \text{exp} ) \]

regexps cannot describe matched ( )
No recursion in definitions

Grammars comprise list of productions
\[ exp \rightarrow \text{id} \]
\[ exp \rightarrow \text{exp} + \text{exp} \]
\[ exp \rightarrow \text{exp} - \text{exp} \]
\[ exp \rightarrow ( \text{exp} ) \]

Left hand side is a non-terminal
Right hand side is sequence of terminals and non-terminals
Terminals are tokens from lexer
One non-terminal is specified as start

A sentence is any complete sequence of terminals in the language defined by the grammar

alternatively

A sentence has complete derivation from the start symbol
Consider starting from \( \text{exp} \) for:
\[
a + b - c
\]

\[
\text{exp} \\
\text{exp} + \text{exp} \\
\text{id} + \text{exp} \\
\text{id} + \text{exp} - \text{exp} \\
\text{id} + \text{id} - \text{exp} \\
\text{id} + \text{id} - \text{id}
\]

Note: *leftmost strategy*
More than one derivation of a given sentence

E.g., we can choose a *rightmost derivation* strategy:
\[
\text{exp} \\
\text{exp} + \text{exp} \\
\text{exp} + \text{exp} - \text{exp} \\
\text{exp} + \text{id} - \text{id} \\
\text{id} + \text{id} - \text{id}
\]

Parse tree represents sets of derivations
Parser enforces language syntax
Parse trees necessary for later compilation

Root is always start symbol
Each derivation step expand a node
Leaves are terminals
exp
exp + exp
id + exp
id + exp - exp
id + id - exp
id + id - id

exp
exp
exp + exp
id + exp
id + exp - exp
id + id - exp
id + id - id

exp
exp
exp + exp
id + exp
id + exp - exp
id + id - exp
id + id - id
Parsing finds a derivation
  of a particular sentence, given a grammar

Most of next 3 weeks:
  Work on ways to construct parsers
  Ways to build parse trees

We deal with realistic (non-trivial) grammars

Grammar may be ambiguous
  Multiple trees for same sentence
  a+b-c

Many languages can be described:
  By ambiguous grammar
  Non-ambiguous grammar
  True for (almost) all interesting grammars

Important points:
  Ambiguous grammars describe language
  Sentence still in or not in (decidable); ambiguity wrt derivation only
Recall that recognizing with NFAs may require backtracking.

Backtracking is inefficient for parsers, unnecessary for programming langs.

Unambiguous grammars yield parsers that don’t require backtracking

Two common ways to eliminate ambiguity:

Precedence
  Provide ordering of productions
  Favor higher precedence rules

Associativity
  Can be left or right
  Prefer deriving rules from left or right

Can also rewrite grammar
Introduce non-terminals to impose precedence

\[\begin{align*}
  \text{exp} &\rightarrow \text{exp} + \text{exp} \\
  \text{exp} &\rightarrow \text{exp} - \text{exp} \\
  \text{id} &\rightarrow \text{id} \\
  (\text{exp}) &\rightarrow (\text{exp}) \\
  \text{exp} &\rightarrow \text{exp} + \text{term} \\
  \text{exp} &\rightarrow \text{term} \\
  \text{term} &\rightarrow \text{term} - \text{factor} \\
  \text{term} &\rightarrow \text{factor} \\
  \text{factor} &\rightarrow \text{id} \\
  \text{factor} &\rightarrow (\text{exp})
\end{align*}\]
How to build parse trees?

Top down- start from root (start state), grow to leaves (non-terminals)

Bottom up- start from non-terminals, reconstruct towards root

Always read tokens left-to-right

Predictive parsers: top-down, expand node with 1-token lookahead

LL(1) grammar: recognizable by predictive parsers
  LL(1): Left-to-right Leftmost lookahead 1

Recursive descent is simplest parsing method

Define one function for each non-terminal

Functions predict path from input token
  productions must vary in first token
Consider simple unambiguous prefix grammar:

\[
\begin{align*}
\text{exp} & \rightarrow + \text{exp} \text{ exp} \\
\text{exp} & \rightarrow - \text{exp} \text{ exp} \\
\text{exp} & \rightarrow \text{term} \\
\text{term} & \rightarrow * \text{term} \text{ term} \\
\text{term} & \rightarrow / \text{term} \text{ term} \\
\text{term} & \rightarrow \text{factor} \\
\text{factor} & \rightarrow \text{id} \\
\text{factor} & \rightarrow \text{num} \\
\text{factor} & \rightarrow ( \text{exp} )
\end{align*}
\]

Function for parsing exp is:

```c
parseExp(Token nextToken)
{
    switch(nextToken.sym)
    {
        case PLUS: parseExp(yylex()); parseExp(yylex()); break;
        case MINUS: parseExp(yylex()); parseExp(yylex()); break;
        default: parseTerm(nextToken); break;
    }
}
```

How do you know if it is LL(1)?

Compute FIRST sets...

Consider sequence of terminals and non-terminals \( \gamma \): FIRST(\( \gamma \)) is set of terminals that can start a sentence expanding \( \gamma \)

Example:

FIRST(exp) = { +, -, *, /, ID, NUM, ( }
If grammar \( g \) has at least
Non-terminal \( X \)
Productions \( X \rightarrow \gamma_1 \) and \( X \rightarrow \gamma_2 \)

Where FIRST(\( \gamma \)) intersects FIRST(\( \gamma_2 \))

Then \( g \) is not LL(1)

---

To compute FIRST sets, compute
FOLLOW and nullable

For any grammar \( g \) and non-terminal \( X \)
FOLLOW(\( X \)) is set of terminals that can follow \( X \)

For any grammar \( g \) and non-terminal \( X \)
nullable(\( X \)) is true iff \( X \) can derive \( \varepsilon \)

---

For grammar:
\[
\begin{align*}
\text{exp} & \rightarrow + \text{exp} \text{exp} \\
\text{exp} & \rightarrow - \text{exp} \text{exp} \\
\text{exp} & \rightarrow \text{term} \\
\text{term} & \rightarrow * \text{term} \text{term} \\
\text{term} & \rightarrow / \text{term} \text{term} \\
\text{factor} & \rightarrow \text{id} \\
\text{factor} & \rightarrow \text{num} \\
\text{factor} & \rightarrow ( \text{exp} ) \\
\end{align*}
\]

FOLLOW(\( \text{factor} \)) = 
\( \{ +, -, *, /, \text{id}, \text{num}, \text{EOF} \} \)
nullable always false for this grammar
FIRST, FOLLOW are smallest sets to satisfy constraints (which also define nullable):

- foreach terminal $T_1 \ldots T_n$
  - nullable($T_i$) = false
- foreach production $X \rightarrow Y_1 Y_2 \ldots Y_k$
  - if foreach $Y_j$ in $Y_1$ to $Y_k$, nullable($Y_j$) = true
  - then nullable($X$) = true
- foreach $i$ in 1 to $k$, foreach $j$ in $i+1$ to $k$
  - if foreach $Y_j$ in $Y_1$ to $Y_i$, nullable($Y_j$) = true
    - then FIRST($Y_j$) subset of FIRST($X$)
  - if foreach $Y_j$ in $Y_i$ to $Y_k$, nullable($Y_j$) = true
    - then FOLLOW($X$) subset of FOLLOW($Y_j$)

FIRST, FOLLOW and nullable can be computed

FIRST, FOLLOW and nullable can define table-driven predictive parser:

- Stack replaces explicit recursion
- Each row describes a non-terminal
- Each column describes a terminal
- Each entry is a production

foreach productions $X \rightarrow \gamma$
  - foreach terminal $T$ in FIRST($\gamma$)
    - enter $X \rightarrow \gamma$ into column $T$ row $X$

foreach productions $X \rightarrow \gamma$
  - if nullable $\gamma$ is true
    - foreach terminal $T$ in FOLLOW($\gamma$)
      - enter $X \rightarrow \gamma$ into column $T$ row $X$
Parsing proceeds by considering symbol X at top of stack and current input a

1. If $X = a = \text{EOF}$, halt successfully
2. If $X = a \neq \text{EOF}$, pop X, advance to next symbol
3. If X nonterminal, replace X on stack with production at $(X, a)$, reversed (error if $(X, a)$ empty)
4. Error o.w.

For the prefix grammar $(e = \text{exp}, t = \text{term}, f = \text{factor})$:

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>ID</th>
<th>NUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>e</td>
<td>+e</td>
<td>-e</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
</tr>
<tr>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td>t</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>ID</td>
<td>ID</td>
<td>ID</td>
<td>ID</td>
<td>ID</td>
<td>ID</td>
<td></td>
</tr>
</tbody>
</table>

Consider parsing $+ a - b c$ as an exp:

<table>
<thead>
<tr>
<th>stack</th>
<th>next token</th>
<th>production</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e$+</td>
<td>+</td>
<td>$+e$</td>
</tr>
<tr>
<td>ee+</td>
<td>+</td>
<td>ID</td>
</tr>
<tr>
<td>et</td>
<td>ID</td>
<td>t</td>
</tr>
<tr>
<td>ef</td>
<td>ID</td>
<td>ID</td>
</tr>
<tr>
<td>$eID$</td>
<td>ID</td>
<td>ID</td>
</tr>
</tbody>
</table>

Now swap to infix language:

- $exp \rightarrow exp + exp$
- $factor \rightarrow id$
- $exp \rightarrow exp - exp$
- $factor \rightarrow num$
- $exp \rightarrow \text{term}$
- $factor \rightarrow ( \text{exp} )$
- $\text{term} \rightarrow \text{term} \ast \text{term}$
- $\text{term} \rightarrow \text{term} / \text{term}$
- $\text{term} \rightarrow \text{factor}$

Ambiguous without associativity

Assume left associative
Can no longer build predictive parser:

```c
parseExp(Token nextToken) {
    switch (nextToken.sym) {
    case ?: parseExp(yylex()); parseExp(yylex());
              break;
    case ?: parseExp(yylex()); parseExp(yylex());
              break;
    default: parseTerm(nextToken);
             break;
    }
}
```

Recursive descent parsing requires LL(1) grammar
non-ambiguous grammar
Limit on grammar form, not language
If X start of rhs for some X prod'n, left-
recursive grammar
Left-recursive grammars never LL(1)

Need to remove all left recursion
accomplished with new non-terminals

For our infix grammar:

- `exp → exp + exp`
- `exp → exp + exp`
- `exp → term`

This grammar LL(1)
Same problem with many statements, e.g. "dangling else":
\[
\text{stmt} \rightarrow \text{if exp then stmt else stmt}
\]
\[
\text{stmt} \rightarrow \text{if exp then stmt}
\]
Consider: if \( \text{exp} \) then if \( \text{exp} \) then stmt else stmt

Can be converted to LL(1):
\[
\text{stmt} \rightarrow \text{if exp then stmt elsepart}
\]
\[
\text{elsepart} \rightarrow \text{else stmt}
\]
\[
\text{elsepart} \rightarrow \epsilon
\]

Left-factoring

Predictive parsers:
Also called predictive
Simplest to write
Recursive or table driven
Recognize LL(1) grammars

Start from expectation
Current non-terminal
Looks for match in input
Predictive parsing is top-down parsing

LL(1) requirement is onerous

We use alternative approach LR(K)
Bottom-up approach
Reconstructs right-most derivation
k token lookahead
LR(K): Left-to-right Rightmost lookahead k

Also called shift-reduce parsing
LR parsing approach

- Push tokens on stack (shift)
- If top of stack matches production, reduce
  - Pop off matched parts
  - Push on non-terminal
- Look ahead character:
  - Helps decide shift or reduce
- Success when stack = start symbol at EOF

Assignment 2
- build lexer for lake

- Yylex.lex in assignment has few rules
  - you must finish them

- Same tokens as C
  - handout has full set described

- Also a problem from the dragon book

Returned tokens instances of LakeSym

- Two useful constructors:
  - LakeSym(LakePos pos, int tokenType)
  - LakeSym(LakePos pos, int tokenType, Object value)

- Values used
  - String for id, string constant
  - Character for char constant
  - Integer for int constant
  - Real for floating constant
Token types defined by jcup
  Become int constants in sym class

Most tokens are obvious
  Assignment operators end in _EQ
  Constant values end in _CONST
  LONG_INT_CONST only for extra credit

Reference as sym.TOKENTYPE for argument to LakeSym

LakePos describes source position
  Starting and ending positions indicated
  Each has source file, line and offset
  Various constructors available

More on using LakePos later

FYI the other classes are:
  Lake main class, error functions
    Run as java lake.Lake filename
  LakeMsg wrapper for a message
  Messages defined messages
    You can add more
  LakeFlags mostly no-op now
  parser, sym cup generated
3 options for extra credit:

1. Fully handle C string semantics
2. Process source line identification lines
3. All integer constant formats

To change file name (for extra credit) set Lake.currentSourceFileName
(yes, a function to set it would be better)

To run Jlex:
> -cdamon/cs202/bin/jlex Yylex.lex

To run Lake:
> setenv CLASSPATH -cdamon/cs202/java:
> java lakec.Lake test1.txt
LPAREN
INT_CONST:1
PLUS
INT_CONST:2
RPAREN