Announcements

- Homework 3 (LLVMlite) out
  - Due Tuesday Oct 15
  - Start early!!!
    - Challenging assignment; HW4 will be released Oct 8
Basic Architecture: Review

- Source Code
  - Parsing
  - Elaboration
  - Lowering
  - Optimization
  - Code Generation
  - Target Code

Front end

Back end
if price > 500 then tax = .08
Today

• **Lexical analysis**: breaks input sequence of characters into individual words, aka “tokens”

if price > 500
then tax = .08
Lexical Tokens

- A language classifies lexical tokens into **token types**

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>foo n14 last</td>
</tr>
<tr>
<td>NUM</td>
<td>73 0 00 515 082</td>
</tr>
<tr>
<td>REAL</td>
<td>66.1 .5 10. 1e67</td>
</tr>
<tr>
<td>IF</td>
<td>if</td>
</tr>
<tr>
<td>COMMA</td>
<td>,</td>
</tr>
<tr>
<td>NOTEQ</td>
<td>!=</td>
</tr>
<tr>
<td>LPAREN</td>
<td>(</td>
</tr>
</tbody>
</table>

- So, a token type specifies a set of acceptable tokens.
- **Reserved words** are tokens that cannot be used as identifiers
  - E.g., IF, VOID, RETURN
Example 1

• Given a program

```plaintext
if (price>500)
    then tax=.08

the lexical analysis returns the sequence of tokens

IF LPAREN ID(price) GT NUM(500) RPAREN THEN
ID(tax) EQ REAL(0.08)
```
Example 2

- Given a program

  ```python
  if (price>500)
  then tax=1xab
  ```

  the lexical analysis returns

  **ERROR**

  because 1xab is neither a number nor an identifier.

- The lexical analysis can help in reporting where an error occurs in the code.
  - By recognizing `\n` as a token and incrementing the line number.

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>foo n14 last</td>
</tr>
<tr>
<td>NUM</td>
<td>73 0 00 515 082</td>
</tr>
<tr>
<td>THEN</td>
<td>then</td>
</tr>
</tbody>
</table>

Stephen Chong, Harvard University
Example 3

• Given a program

\[
\text{if (price}>500) \\
\text{then tax} = .08
\]

the lexical analysis returns

\[
\text{IF LPAREN ID(price) GT NUM(500) RPAREN ID(thn) ID(tax) EQ REAL(0.08)}
\]

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>foo n14 last</td>
</tr>
<tr>
<td>NUM</td>
<td>73 0 00 515 082</td>
</tr>
<tr>
<td>THEN</td>
<td>then</td>
</tr>
</tbody>
</table>

• Is this an error at the level of lexical analysis?
  • No, it is an error at the level of syntax analysis (next lectures)!
Towards Implementing A Lexical Analysis

- Recall: Lexical analysis breaks input into tokens.
- The lexical analysis needs to decide the token type for a given string (i.e., sequence of characters).

Is it a:
- ID ?
- NUM ?
- REAL?
- IF ?
- LPAREN ?
...

String \[\rightarrow\] Token type
Let’s simplify…

• Recall: Lexical analysis breaks input into tokens.
• The lexical analysis needs to decide the token type for a given string (i.e., sequence of characters).
A Set Membership Question

• Recall: a token type specifies a set of acceptable tokens (i.e., strings).
  • The set of acceptable tokens for NUM is \{0,1,2,3,…\}.
• But this set is infinite…

Does it belong to:
- \{0,1,2,3,…\} ?
A Set Membership Question

• How can we mechanically decide if a string belongs to a (possibly infinite) set $S$ of strings?

• An approach:
  • Use a finite representation of $S$.
    ‣ Regular expressions
  • Check whether the string is accepted by such a finite representation.
    ‣ Deterministic finite-state automata
Regular Expressions

• Each regular expression represents a set of strings.

• Examples
  • \((0 \mid 1)^* 0\)
    • Binary numbers that are multiples of 2
  • \(b^*(abb^*)^*(a \mid \varepsilon)\)
    • Strings of a’s and b’s without consecutive a’s
  • \((a \mid b)^*aa(a \mid b)^*\)
    • Strings of a’s and b’s with consecutive a’s
Regular Expressions (RE)

• Grammar
  • $\emptyset$ (matches no string)
  • $\varepsilon$ (epsilon – matches empty string)
  • Literals (‘a’, ‘b’, ‘2’, ‘+’, etc.) drawn from alphabet
  • Concatenation ($R_1 R_2$)
  • Alternation ($R_1 \mid R_2$)
  • Kleene star ($R^*$)
Set of Strings

- $\begin{array}{c} \left[ \emptyset \right] = \{ \} \\ \left[ \epsilon \right] = \{ \"\" \} \\ \left[ \text{‘a’} \right] = \{ \"a\" \} \\ \left[ R_1 \; R_2 \right] = \{ s \mid s = \alpha \wedge \beta \text{ and } \alpha \in \left[R_1\right] \text{ and } \beta \in \left[R_2\right] \} \\ \left[ R_1 \; | \; R_2 \right] = \{ s \mid s \in \left[R_1\right] \text{ or } s \in \left[R_2\right] \} \\ = \left[R_1\right] \cup \left[R_2\right] \\ \left[ R^* \right] = \left[ \epsilon \; | \; RR^* \right] \\ = \{ s \mid s = \"\" \text{ or } s = \alpha \wedge \beta \text{ and } \alpha \in \left[R\right] \text{ and } \beta \in \left[R^*\right] \} \end{array}$
Syntactic Sugar

- [0-9] shorthand for 0 | 1 | … | 9
- R? shorthand for (R | ε) (i.e., R is optional)
- R+ shorthand for (R R*) (i.e., at least one R)
### Regular Expressions to Specify Token Types!

<table>
<thead>
<tr>
<th>Reg Exp</th>
<th>Token Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>if</td>
<td>IF</td>
</tr>
<tr>
<td>[a-z][a-z0-9]</td>
<td>ID</td>
</tr>
<tr>
<td>[0-9]+</td>
<td>NUM</td>
</tr>
<tr>
<td>([0-9]+ &quot;.&quot; [0-9]*)</td>
<td></td>
</tr>
</tbody>
</table>

- **Question:** What is the token type of input `iffy`?
  - We want the token `ID(iffy)` rather than `IF`.

- **In general, we want the longest match:**
  - longest initial substring of the input that can match a regular expression is taken as next token.
Lexical analysis breaks input into tokens.
The lexical analysis needs to decide the token type for a given string (i.e., sequence of characters).
A Matching Question

- Lexical analysis breaks input into tokens.
- The lexical analysis needs to decide the token type for a given string (i.e., sequence of characters).

Does it match - [0-9]+ ?

String → Does it match - [0-9]+ ? → Token type
A Deterministic Finite-state Automaton (DFA) can be used to decide whether an input matches a regular expression.

Example: DFA for regular expression \([0-9]+\) :
Other DFAs

- **IF**
  - Start state
  - Transitions: 
i → accept
  - Accept state

- **ID**
  - Start state
  - Transitions: 
a-z → accept
  - Accept state
  - Additional transition: 
    a-z → ID

- **REAL**
  - Start state
  - Transitions: 
    0-9 → accept
    0-9 → REAL
    . → accept
    0-9 → REAL
  - Accept state
This DFA takes as an input a sequence of characters and returns a Token Type (if the input is accepted).

So, this DFA can be used for Lexical Analysis.
Using DFAs

- Usually record transition function as array indexed by state and characters (i.e., transition table)
  - See Appel Chap 2.3 for an example.
How is a RE converted to a DFA?

1. Convert RE to a Nondeterministic Finite-state Automaton (NFA).
2. Convert NFA to DFA.
RE to NFA conversion

- **Epsilon ε**
  - start \(\xrightarrow{\varepsilon}\) accept

- **Literal ‘a’**
  - start \(\xrightarrow{a}\) accept

- **Concatenation \(R_1R_2\)**

- **Alternation \(R_1 \mid R_2\)**
RE to NFA conversion

• Kleene star $R^*$
NFA to DFA conversion (intuition)

• The NFA of a regular expression \( R \) can be easily composed from NFAs of subexpressions of \( R \).
• But executing an NFA under input strings is harder and less efficient than executing a DFA due to the nondeterminism.
• So, we convert NFAs to DFAs.
  • Basic idea: each state in DFA will represent a set of states of the NFA.
Example: NFA to DFA

NFA:

DFA:
Example: NFA to DFA

NFA:

DFA:
Example: NFA to DFA

NFA:

DFA:

Check that this DFA is, in fact, deterministic!
Lexical Analysis Summary

• Use a regular expression $R_i$ to specify the set strings for each Token Type.
  • Example: $[0-9]^+$ specifies the set of strings for NUM

• Construct the NFA formed by $(R_1 | R_2 | \ldots | R_n)$.

• Construct the DFA for this NFA.

• Produce the transition table for that DFA.

• Implement longest match.
Using a Lexer Generator

• The designer of a lexical analysis follows the first step of the previous slide.
• The remaining steps are automatically performed by the lexer generator!
A Lexer Generator in ML

• Provide regular expressions for token types in file mllexeg.mll
• Run lexer generator: `ocamllex mllexeg.mll`
• The lever generator produces the final transition table at file mllexeg.ml
Structure of ocamllex File

{ header }
let ident = regexp ...
rule entrypoint1 [arg1 ... argn] =
  parse regexp { action }
  | ...
  | regexp { action }
and entrypoint2 [arg1 ... argn] =
  parse ...
and ...
{ trailer }

- Header and trailer are arbitrary OCaml code, copied to the output file
- Can define abbreviations for common regular expressions
- Rules are turned into (mutually recursive) functions with args1 ... argn lexbuf
  - lexbuf is of type Lexing.lexbuf
  - Result of function is the result of ml code action
A hand-written Lexer

- See file lexer.ml