Motivation...
Big picture

- Front end responsibilities
  - Check that the input program is legal
    - Check syntax and semantics
    - Emit meaningful error messages
  - Build IR of the code for the rest of the compiler

Front end

- Strategy:
  - Specify the input language formally
    *Regular expressions, context-free grammars*
  - Use automatons to recognize valid strings
    *Hey, this stuff is actually useful?*
  - Automate construction
    *Or, in our case, learn how automation works*
  - Add “actions” to state transitions
    - Build IR data structure
    - Perform additional semantic checks
Front end design

- Two part design
  - Scanner
    - Reads in characters
    - Classifies sequences into words or tokens
  - Parser
    - Checks sequence of tokens against grammar
    - Creates a representation of the program (AST)

Scanner and parser

- Why separate scanner and parser?
  - Simplifies the implementation
  - Parsing is fundamentally harder
    - Word classification is easier – make it fast
    - Speed up parsing by working with tokens
Scanner

- Responsibilities
  - Read in characters
  - Produce a stream of tokens
    \[ \text{key,if} \{ x, -, 5 \} \{ \ldots \} \]
    \[ \text{id} \{ \ldots \} \text{op,==} \text{num,5} \{ \ldots \} \]
  - Token has a type and a value

Scanner

- Relation to parser

1. \( \text{goal} \rightarrow \text{expr} \)
2. \( \text{expr} \rightarrow \text{expr op term} \)
3. \( \text{term} \rightarrow \text{number} \)
4. \( \text{term} \rightarrow \text{id} \)
5. \( \text{op} \rightarrow + \)
6. \( \text{op} \rightarrow - \)

- Could be encoded in grammar:

6. \( \text{number} \rightarrow \text{number digit} \)
7. \( \text{digit} \rightarrow \text{digit} \)
8. \( \text{digit} \rightarrow 0 \mid 1 \mid 2 \mid 3 \ldots \)
Hand-coded scanner

- Explicit test for each token
- Read in a character at a time
- Example: recognizing keyword “if”

```c
char c = readchar();
if (c != 'i')
    error();
else {
    c = readchar();
    if (c != 'f')
        error();
    else
        return IF_TOKEN;
}
```

Scanner construction

- **Goal**: automate process
- Avoid writing scanners by hand
- Leverage the underlying theory of languages
Scanner construction

- Tokens specified as **regular expressions**
  
  *Note: in PL, spelling identifies part of speech*
  
- Scanner generator produces state machine
  - Recognizes the REs
  - Implemented as tables or directly in code

---

Regular expressions

- Rules or patterns to define **regular languages**
  - Alphabet $\Sigma$
  - Language is a set of strings
  - Let $L(r)$ denote the language described by RE $r$

- Regular expressions over $\Sigma$
  - $\epsilon$ is an RE denoting empty set
  - if $a$ is in $\Sigma$, then $a$ is an RE for $\{a\}$
  - if $x$ and $y$ are REs then:
    - $xy$ is an RE for $L(x)L(y)$ **Concatenation**
    - $x|y$ is an RE for $L(x) \cup L(y)$ **Alternation**
    - $x^*$ is an RE for $L(x)^*$ **Kleene closure**
Set operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union of $L$ and $M$</td>
<td>(L \cup M = {s \mid s \in L \text{ or } s \in M})</td>
</tr>
<tr>
<td>Concatenation of $L$ and $M$</td>
<td>(LM = {st \mid s \in L \text{ and } t \in M})</td>
</tr>
<tr>
<td>Kleene closure of $L$</td>
<td>(L^* = \bigcup_{0 \leq i \leq \infty} L^i)</td>
</tr>
<tr>
<td>Positive Closure of $L$</td>
<td>(L^+ = \bigcup_{1 \leq i \leq \infty} L^i)</td>
</tr>
</tbody>
</table>

Using regular expressions

- Concatenation: build up words
- Kleene closure: repetition
- Alternation: collect sets of words
Examples

Identifiers:
- Letter → (a|b|c| ... |z|A|B|C| ... |Z)
- Digit → (0|1|2| ... |9)
- Identifier → Letter ( Letter | Digit )

Numbers:
- Integer → (+|-|ε) (0|1|2| ... |9)(Digit )
- Decimal → Integer  Digit
- Real → ( Integer | Decimal ) E (+|-|ε) Digit
- Complex → ( Real , Real )

Numbers can get much more complicated!

Back to scanners

- How do we use regular expressions?
  - Every RE has an equivalent finite state automaton that recognizes its language
    (Actually, more than one)
  - Example: a(b|c)*

- Idea: scanner simulates the automaton
Example

- Consider the problem of recognizing register names

\[ Register \rightarrow r \ 012 \ldots 9 \ 012 \ldots 9^* \]

![Diagram showing state transitions for recognizing register names]

- Start in state \( S_0 \) & take transitions on each input character
- FA accepts a word \( x \) iff \( x \) leaves it in a final state \( (S_2) \)
- Other transition go to an error state, \( S_e \)

Implementation

- Finite automaton
  - States, characters
  - State transition \( \delta(\text{state},\text{charclass}) \) determines next state
  - Automaton is deterministic

- Next character function
  - Reads next character into buffer
  - (May compute character class by fast table lookup)

- Transitions from state to state
  - Implement \( \delta \) as a table
  - Access table using current state and character
### Example

#### Turning the recognizer into code

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>$r$</th>
<th>$0,1,2,3,4,$</th>
<th>All others</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$5,6,7,8,9$</td>
<td></td>
</tr>
<tr>
<td>$s_0$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
</tbody>
</table>

Char $\leftarrow$ next character  
State $\leftarrow s_0$

while (Char $\neq$ EOF)  
State $\leftarrow \delta$(State, Char)  
Char $\leftarrow$ next character

if (State is a final state)  
then report success  
else report failure

---

#### Example

#### Adding actions

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>$r$</th>
<th>$0,1,2,3,4,$</th>
<th>All others</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$5,6,7,8,9$</td>
<td></td>
</tr>
<tr>
<td>$s_0$</td>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td></td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
</tbody>
</table>

Char $\leftarrow$ next character  
State $\leftarrow s_0$

while (Char $\neq$ EOF)  
State $\leftarrow \delta$(State, Char)  
perform specified action  
Char $\leftarrow$ next character

if (State is a final state)  
then report success  
else report failure
What if we need a tighter specification?

- \( r \ Digit \ Digit \) allows arbitrary numbers
  - Accepts \( r00000 \)
  - Accepts \( r99999 \)
  - What if we want to limit it to \( r0 \) through \( r31 \) ?

- Write a tighter regular expression
  - \( \text{Register} \rightarrow r ( (0|1|2) (Digit \ | \epsilon) | (4|5|6|7|8|9) | (3|30|31) ) \)
  - \( \text{Register} \rightarrow r0|r1|r2 \ldots |r31|r00|r01|r02 \ldots |r09 \)

- Produces a more complex DFA
  - Has more states
  - Same cost per transition
  - Same basic implementation

Tighter register specification

- The DFA for
  \[
  \text{Register} \rightarrow r ( (0|1|2) (Digit \ | \epsilon) | (4|5|6|7|8|9) | (3|30|31) )
  \]

- Accepts a more constrained set of registers
- Same set of actions, more states
Tighter register specification

<table>
<thead>
<tr>
<th>$\delta$</th>
<th>$r$</th>
<th>0,1</th>
<th>2</th>
<th>3</th>
<th>4-9</th>
<th>All others</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_0$</td>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_1$</td>
<td>$s_e$</td>
<td>$s_2$</td>
<td>$s_2$</td>
<td>$s_5$</td>
<td>$s_4$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$s_e$</td>
<td>$s_3$</td>
<td>$s_3$</td>
<td>$s_3$</td>
<td>$s_3$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_3$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_4$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_5$</td>
<td>$s_e$</td>
<td>$s_6$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_6$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
<tr>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
<td>$s_e$</td>
</tr>
</tbody>
</table>

Table encoding RE for the tighter register specification

Building DFAs

- Each RE has an equivalent DFA
  - May be hard to directly construct the right DFA

- What about an RE such as $(a | b)^* abb$?

- This FA is different:
  - $S_0$ has a transition on $\epsilon$
  - $S_1$ has two transitions on $a$

- This is a non-deterministic finite automaton

Deterministic automaton
Non-deterministic finite automata

- An NFA accepts a string $x$ iff $\exists$ a path though the transition graph from $s_0$ to a final state such that the edge labels spell $x$.
  \[\text{\text{(Transitions on } \varepsilon \text{ consume no input)}\]

- To “run” the NFA, start in $s_0$ and guess the right transition at each step.
  - Always guess correctly
  - If some sequence of correct guesses accepts $x$ then accept

- Why study NFAs?
  - They are the key to automating the RE $\rightarrow$ DFA construction
  - (We can paste together NFAs with $\varepsilon$-transitions)

NFA Example

- Input:
  - Must know the future

- Input:
  - $abaab$
Relationship between NFAs and DFAs

- DFA is a special case of an NFA
  - DFA has no $\varepsilon$ transitions
  - DFA’s transition function is single-valued
  - Same rules will work
- DFA can be simulated with an NFA \textit{(obvious)}
- NFA can be simulated with a DFA \textit{(less obvious)}
  - Simulate sets of possible states
  - Possible exponential blowup in the state space
  - Still, one state per character in the input stream

Automatic scanner construction

- To convert a specification into code:
  1. Write down the RE for the input language
  2. Build a big NFA
  3. Build the DFA that simulates the NFA
  4. Systematically shrink the DFA
  5. Turn it into code

- Scanner generators
  - Lex and Flex work along these lines
  - Algorithms are well-known and well-understood
  - Key issue is interface to parser \textit{(define all parts of speech)}
  - You could build one in a weekend!
Automatic scanner construction

RE $\rightarrow$ NFA  (*Thompson's construction*)
- Build an NFA for each term
- Combine them with $\varepsilon$-moves

NFA $\rightarrow$ DFA (*subset construction*)
- Build the simulation

DFA $\rightarrow$ Minimal DFA
- Hopcroft's algorithm

DFA $\rightarrow$ RE  (*Not part of the scanner construction*)
- All pairs, all paths problem
- Take the union of all paths from $s_0$ to an accepting state

C scanner

```c
{%
#include "c_breeze.h"
#include "parser.tab.h"
%
identifier       ([a-zA-Z_]\[0-9a-zA-Z_\]*)
short_escape     ([0-7]["'\n\"]\)
any_white         ([\011\013\014\015\ ])%

{any_white}+     { }
for
  { lval.tok = get_pos(); return ctokFOR; }
if
  { lval.tok = get_pos(); return ctokIF; }
{identifier}
  { lval.tok = get_pos();
    lval.idN = new idNode(cbtext, cblval.tok);
    if ( is_typename(cbtext)) return TYPEDEFname;
    else return IDENTIFIER; }
{decimal_constant}
  { lval.expN = atoi(cbtext);
    return INTEGERconstant; }
%
...any special code...
```
**Next time...**

- Sign up on mailing list:
  https://www.eecs.tufts.edu/mailman/listinfo/comp181

- RE → NFA → DFA → scanner

- Algorithms (yikes!)
Automatic scanner construction

- Construct a DFA to recognize any RE
- Overview:
  - Direct construction of a nondeterministic finite automaton (NFA) to recognize a given RE
  - Construct a deterministic finite automaton (DFA) to simulate the NFA
  - Minimize the number of states
  - Generate the scanner code