Prelude

"God does not play dice with the universe."

What does this quote refer to?

- Newtonian physics: nature is **deterministic**
- Quantum mechanics: nature is **non-deterministic**
- Einstein didn’t like this
- We will take Einstein’s view…

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 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)

Lexical analysis

- The input is just a sequence of characters.
  - Example:
    ```
    if (i == j) {
      z = 0;
    } else
      z = 1;
    ```
- More accurately, the input is:
  ```
  \tif (i == j)\n  | n\n  | tz = 0;\n  | \n  | else\n  | n\n  | tz = 1;
  ```
- **Goal**: Partition input string into substrings
  - And classify them according to their role

Scanner

- Responsibilities
  - Read in characters
  - Produce a stream of **tokens**

  ```
  <key,if> <id,i> <op,==> <id,j> ...
  ```
Hand-coded scanner

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

- What about other tokens?
  - Example: "if" is a keyword, "if0" is an identifier

```
c = readchar();
if (c != 'i') {
  other tokens...
} else {
  c = readchar();
  if (c != 'f') {
    other tokens...
  } else {
    c = readchar();
    if (c not alpha-numeric) {
      putback(c);
      return IF_TOKEN;
    } while (c alpha-numeric) {
      build identifier
    }
  }
}
```

Hand-coded scanner

Problems:
- Many different kinds of tokens
  - Fixed strings (keywords)
  - Special character sequences (operators)
  - Tokens defined by rules (identifiers, numbers)
- Tokens overlap
  - "if" and "if0" example
  - "=' and "=="
- Coding this by hand is too painful!
  - Getting it right is a serious concern

Scanner construction

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

```
Source code

Scanner

Generator

Compile time

Design time

Specification

Scanner

Generator

tokens
```

Outline

Problems we need to solve:
- Scanner description
  - How to describe parts of the input language
- The scanning mechanism
  - How to break input string into tokens
- Scanner generator
  - How to translate from (1) to (2)
- Ambiguities
  - The need for lookahead

Problem 1: Describing the scanner

- We want a high-level language D that
  1. Describes lexical components, and
  2. Maps them to tokens (determines type)
- **But** doesn't describe the scanner algorithm itself!
- Part 3 is important
  - Allows focusing on what, not on how
- Therefore, D is sometimes called a **specification language**, not a programming language
- Part 2 is easy, so let's focus on Parts 1 and 3
Specifying tokens

- Many ways to specify them
- **Regular expressions** are the most popular
  - REs are a way to specify sets of strings
  - Examples:
    - \( a \) – denotes the set \{ \( a \) \}
    - \( ab \) – denotes the set \{ \( a \), \( b \) \}
    - \( ab^* \) – denotes the set \{ \( a \), \( ab \), \( abb \), \( abbb \), … \}
- Why regular expressions?
  - Easy to understand
  - Strong underlying theory
  - Very efficient implementation

Formal languages

- **Def:** a language is a set of strings
  - Alphabet \( \Sigma \) : the character set
  - Language is a set of strings over alphabet
- Each regular expression denotes a language
  - If \( A \) is a regular expression, then \( L(A) \) is the set of strings denoted by \( A \)
  - Examples: given \( \Sigma = \{ \text{'a', 'b'} \} \)
    - \( A = \text{'a'} \) \( \Rightarrow L(A) = \{ \text{'a'} \} \)
    - \( A = \text{'a'} \mid \text{'b'} \) \( \Rightarrow L(A) = \{ \text{'a'}, \text{'b'} \} \)
    - \( A = \text{'a'} \text{'b'} \) \( \Rightarrow L(A) = \{ \text{'ab'} \} \)
    - \( A = \text{'a'} \text{'b'}^* \) \( \Rightarrow L(A) = \{ \text{'a'}, \text{'ab'}, \text{'abb'}, \text{'abbb'}, \ldots \} \)

Building REs

- Regular expressions over \( \Sigma \)
  - Atomic REs
    - \( \varepsilon \) is an RE denoting empty set
    - if \( a \) is in \( \Sigma \), then \( a \) is an RE for \{ \( a \) \}
  - Compound REs
    - 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

Examples

- In class…

Outline

Problems we need to solve:
- Scanner specification language **DONE**
  - How to describe parts of the input language
- The scanning mechanism
  - How to break input string into tokens
- Scanner generator
  - How to translate from (1) to (2)
- Ambiguities
  - The need for lookahead

Overview of scanning

- How do we recognize strings in the language?
  - *Every RE has an equivalent finite state automaton that recognizes its language.*
    - (Often more than one)
  - **Idea:** scanner simulates the automaton
    - Read characters
    - Transition automation
    - Return a token if automaton accepts the string
Finite Automata

- Regular expressions = specification
- Finite automata = implementation

A finite automaton consists of:
- An input alphabet Σ
- A set of states S
- A start state s₀
- A set of accepting states F ⊆ S
- A set of transitions state → input state

Finite Automata State Graphs

- A state
- The start state
- An accepting state
- A transition

FA Example

- Transition S₁ → a S₂
- Is read In state s₁ on input "a" go to state s₂
- FA accepts a string if we can follow transitions labeled with the characters in the string from the start to an accepting state
- What if we run out of characters?
- A finite automaton that accepts only "1"

Another Simple Example

- FA accepts any number of 1’s followed by a single 0
- Alphabet: {0, 1}
- Check that “1110” is accepted but “1101…” is not

And Another Example

- Alphabet {0, 1}
- What language does this recognize?

“Realistic” example

- Recognizing machine register names
- Typically “r” followed by register number (how many?)

Recognizer for Register takes it through s₀, s₁, s₂, and accepts.
Recognizer for Register takes it through s₀, s₁, and fails.
Recognizer for Register takes it straight to s₂.
REs and DFAs

- **Key idea:**
  - Every regular expression has an equivalent DFA that accepts only strings in the language

- **Problem:**
  - How do we construct the DFA for an arbitrary regular expression?
  - Not always easy

Example

- What is the FA for a(a|ε)b?
- Need ε moves
- Transition A to B without consuming input!

Another example

- Remember this DFA?
- We can simplify it as follows:

A different kind of automaton

- Accepts the same language
  - Actually, it's easier to understand!
- What's different about it?
  - Two different transitions on '0'
  - This is a **non-deterministic finite automaton**

DFAs and NFAs

- **Deterministic Finite Automata (DFA)**
  - One transition per input per state
  - No ε-moves

- **Nondeterministic Finite Automata (NFA)**
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves

Execution of Finite Automata

- DFA can take only one path through the state graph
  - Completely determined by input

- NFAs can choose
  - Whether to make ε-moves
  - Which of multiple transitions for a single input to take
Acceptance of NFAs

- An NFA can get into multiple states

[Diagram of an NFA with multiple states connected by transitions]

- **Input:**
- **Rule:** NFA accepts if it can get in a final state

Non-deterministic finite automata

- An NFA accepts a string $x$ if there is a path through the transition graph from $s_0$ to a final state such that the edge labels spell $x$
  (Transitions on $\varepsilon$ 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 do we care about NFAs?

- Simpler, smaller than DFAs
- More importantly:
  - Need them to support all RE capabilities
  - Systematic conversion from REs to NFAs
  - Need $\varepsilon$ transitions to connect RE parts
- **Problem:** how to implement NFAs?
  - How do we guess the right transition?

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 **(obvious)**
- NFA can be simulated with a DFA **(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
  - You could build one in a weekend!

Scanner construction

- Define tokens as regular expressions
- Construct NFA for all REs
  - Connect REs with $\varepsilon$ transitions
  - Thompson’s construction
- Convert NFA into a DFA
  - DFA is a simulation of NFA
  - Possibly much larger than NFA
  - Subset construction
- Minimize the DFA
  - Hopcroft’s algorithm
- Generate implementation
[1] Thompson’s construction

- **Goal:**
  Systematically convert regular expressions for our language into a finite state automaton

- **Key idea:**
  - FA “pattern” for each RE operator
  - Start with atomic REs, build up a big NFA
  - Idea due to Ken Thompson in 1968

Thompson’s construction

By induction on RE structure

- **Base case:**
  Construct FA that recognizes atomic regular expressions:

- **Induction:**
  Given FAs for two regular expressions, x and y, build a new FA that recognizes:
  - xy
  - x|y
  - x^*

Need for ε transitions

- What if x and y look like this:
- Then xy ends up like this:

Example

Regular expression: a (b | c)^*

- a, b, & c
- b | c
- (b | c)^*
Example

- \((b|c)^*\)

- Note: a human could design something simpler…
  - Like what?

Problem

- How to implement NFA scanner code?
  - Will the table-driven scheme work?
  - Non-determinism is a problem
  - Explore all possible paths?

- Observation:
  - We can build a DFA that simulates the NFA
    - Accepts the same language
    - Explores all paths simultaneously

[2] NFA to DFA

- Subset construction algorithm
  - Intuition: each DFA state represents the possible states reachable after one input in the NFA

- Two key functions
  - \(\text{next}(s, \alpha)\) – the set of states reachable from \(s\) on \(\alpha\)
  - \(\epsilon\)-closure\((s)\) – the set of states reachable from \(s\) on \(\epsilon\)

- DFA transition function
  - Edge labeled \(a\) from state \(s\) to state \(\epsilon\)-closure\(\text{next}(s, a)\)

NFA to DFA example

- Convert each subset in \(S\) into a state:

  All transitions are deterministic
  - Smaller than NFA, but still bigger than necessary

Subset construction

- Algorithm
  - Build a set of subsets of NFA states

  Initialize the worklist to this one subset

  While there are more subsets on the worklist, remove the next subset

  Apply each input symbol to the set of NFA states to produce a new set.

  If we haven’t seen that subset before, add it to \(S\) and the worklist, and record the set-to-set transition
Does it work?
- Does the algorithm halt?
  - $S$ contains no duplicate subsets
  - $2^{|NFA|}$ is finite
  - Main loop adds to $S$, but does not remove
    it is a monotone function
- $S$ contains all the reachable NFA states
  - Tries all input symbols, builds all NFA configurations
- **Note:** important class of compiler algorithms
  - Fixed point computation
  - Monotonic update function
  - Convergence is guaranteed

[3] DFA minimization
- Hopcroft's algorithm
  - Discover sets of equivalent states in DFA
  - Represent each set with a single state
- When would two states in the DFA be equivalent?
  - Two states are equivalent **iff**
    - The set of paths leading to them are the same
    - For all input symbols, transitions lead to equivalent states
  - This is the key to the algorithm

DFA minimization
- A partition $P$ of the states $S$
  - Each $s \in S$ is in exactly one set $p_i \in P$
  - Idea:
    - If two states $s$ and $t$ transition to different partitions, then they must be in different partitions
- Algorithm:
  - Iteratively partition the DFA’s states
  - Group states into maximal size sets, optimistically
  - Iteratively subdivide those sets, as needed
  - States that remain grouped together are equivalent

Splitting $S$ around $\alpha$
- Original set $S$
- $S_2$ is everything in $S - S_1$
  - Could we split $S_2$ further?
  - Yes, but it does not help asymptotically

DFA minimization
- Details:
  - Given DFA $(S, \Sigma, \delta, s_0, F)$
  - Initial partition: $P_0 = \{F, S - F\}$
  - Intuition: final states are always different
- Splitting a set around symbol $a$
  - Assume $s_a, s \in P_i$, and $\delta(s, a) = s_a$ & $\delta(s, a) = s$
  - Split $P_i$:
    - If $s_a$ & $s$ are not in the same set
    - If $s_a$ has a transition on $a$, but $s$ does not
      - Intuition: one state in DFA cannot have two transitions on $a$
DFA minimization algorithm

\[ P = \{ F, \{Q - F\}\} \]
while (P is still changing)
for each set \( S \in P \)
for each \( \alpha \in \Sigma \) partition \( S \) by \( \alpha \) into \( S_1 \) and \( S_2 \)
if \( T \neq P \) then
\( P \leftarrow T \)

This is a fixed-point algorithm!

Does it work?

- Algorithm halts
- Partition \( P \in 2^S \)
- Start off with 2 subsets of \( S \) (F) and (S-F)
- While loop takes \( P \rightarrow P_{i+1} \) by splitting 1 or more sets
- \( P_{i+1} \) is at least one step closer to partition with |S| sets
- Maximum of |S| splits

Note that
- Partitions are never combined
- Initial partition ensures that final states are intact

Refining the algorithm

- As written, it examines every \( S \in P \) on each iteration
  - This does a lot of unnecessary work
  - Only need to examine \( S \) if some \( T \), reachable from \( S \), has been split
- Reformulate the algorithm using a "worklist"
  - Start worklist with initial partition, \( F \) and \( \{Q - F\} \)
  - When it splits \( S \) into \( S_1 \) and \( S_2 \), place \( S_2 \) on worklist

This version looks at each \( S \in P \) many fewer times

Well-known, widely used algorithm due to John Hopcroft

Hopcroft's algorithm

\[ W = \{ F, \{Q - F\}\} P = \{ F, \{Q - F\}\} \]

while (W is not empty)
do begin
select and remove \( S \) from \( W \);
for each \( \alpha \) in \( \Sigma \) do begin
let \( I_\alpha = \delta^{-1}_\alpha(S) \);
for each \( R \in P \) such that \( R \cap I_\alpha \) is not empty and \( R \) is not contained in \( I_\alpha \) do begin
partition \( R \) into \( R_1 \) and \( R_2 \) such that \( R_1 = R \cap I_\alpha \) and \( R_2 = R - R_1 \);
replace \( R \) in \( P \) with \( R_1 \) and \( R_2 \);
end
end
end

Full example

- Consider \( (a | b)^* \)abb
- Applying the subset construction:

<table>
<thead>
<tr>
<th>Iter.</th>
<th>State</th>
<th>Contains</th>
<th>( r)-closure( moves, ( a ))</th>
<th>( r)-closure( moves, ( b ))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( a_0 )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>1</td>
<td>( a_1 )</td>
<td>( {a_0} )</td>
<td>( {a_0} )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>2</td>
<td>( a_2 )</td>
<td>( {a_0, a_1} )</td>
<td>( {a_0, a_1} )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>3</td>
<td>( a_3 )</td>
<td>( {a_0, a_1, a_2} )</td>
<td>( {a_0, a_1, a_2} )</td>
<td>( \emptyset )</td>
</tr>
</tbody>
</table>

contains \( a_2 \) (final state)

- Iteration 3 adds nothing to \( S \), so the algorithm halts

Full example

- DFA for \( (a | b)^* \)abb

About the same size as NFA
Full example – minimization

| Current Partition | Worklist | | | Split on | Split at | |
|-------------------|----------|------------|----------|------------|----------|
| P₁ (4a (6a, 6b, 6c)) | (4a) | (6a) | none | (6a, 6b, 6c) | (6a) |
| P₂ (6a (6a, 6b, 6c)) | (6a) | (6a) | none | (6a, 6b, 6c) | (6a) |
| P₃ (6a, 6b, 6c) | (6a, 6b, 6c) | (6a, 6b, 6c) | none | (6a, 6b, 6c) | (6a, 6b, 6c) |

Another example

- What about sₙ (b | c)⁺

We’re done with the theory

- Take a nice, deep breath…

Implementation

- Finite automaton
  - States, characters
  - State transition δ uniquely determines next state
- Next character function
  - Reads next character into buffer
  - (May compute character class by fast table lookup)
- Transitions from state to state
  - Implement δ as a table
  - Access table using current state and character

Example

Turning the recognizer into code

Table encoding RE

<table>
<thead>
<tr>
<th>δ</th>
<th>r</th>
<th>0.1.2.3.4.5</th>
<th>0.6.7.8.9</th>
<th>All others</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₁</td>
<td>s₁</td>
<td>s₁</td>
<td>s₁</td>
<td></td>
</tr>
<tr>
<td>s₂</td>
<td>s₂</td>
<td>s₂</td>
<td>s₂</td>
<td></td>
</tr>
<tr>
<td>s₃</td>
<td>s₃</td>
<td>s₃</td>
<td>s₃</td>
<td></td>
</tr>
<tr>
<td>s₄</td>
<td>s₄</td>
<td>s₄</td>
<td>s₄</td>
<td></td>
</tr>
</tbody>
</table>

Char -- next character
State -- s₁
while (Char ≠ EOF)
State = δ(State, Char)
Char -- next character
if (State is a final state)
then report success
else report failure

Skeleton recognizer

Example

Adding actions

Table encoding RE

<table>
<thead>
<tr>
<th>δ</th>
<th>r</th>
<th>0.1.2.3.4.5</th>
<th>0.6.7.8.9</th>
<th>All others</th>
</tr>
</thead>
<tbody>
<tr>
<td>s₁</td>
<td>s₁ start</td>
<td>s₁ error</td>
<td>s₁ end</td>
<td></td>
</tr>
<tr>
<td>s₂</td>
<td>s₂ error</td>
<td>s₂ add</td>
<td>s₂ error</td>
<td></td>
</tr>
<tr>
<td>s₃</td>
<td>s₃ error</td>
<td>s₃ add</td>
<td>s₃ error</td>
<td></td>
</tr>
</tbody>
</table>

Char -- next character
State -- s₁
while (Char ≠ EOF)
State = δ(State, Char)
perform specified action
Char -- next character
if (State is a final state)
then report success
else report failure

Skeleton recognizer
Tighter register specification

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

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

Building a scanner

- Language: if | while | [a-zA-Z][a-zA-Z0-9]* | [0-9][0-9]* 

  - Problem:
    - Giant NFA either accepts or rejects a one token
    - We need to partition a string, and indicate the kind

Partitioning

- Input: stream of characters
- Giant NFA

  - Annotate the NFA
  - Remember the accepting state of each RE
  - Annotate with the kind of token
  - Does giant NFA accept some substring \( x_0 \ldots x_i \)?
  - Return substring and kind of token
  - Restart the NFA at \( x_{i+1} \)

Partitioning problems

- Matching is ambiguous
  - Example: "foo+3"
  - We want <foo>, <+>, <3>
  - But: <f>, <oo>, <+>, <3> also works with our NFA
    - Can end the identifier anywhere
    - Note: "foo+" does not satisfy NFA
  - Solution: "maximal munch"
    - Choose the longest substring that is accepted
    - Must look at the next character to decide -- lookahead
    - Keep munching until no transition on lookahead

More problems

- Some strings satisfy multiple REs
  - Example: "new foo"
    - <new> could be an identifier or a keyword
  - Solution: rank the REs
    - First, use maximal munch
    - Second, if substring satisfies two REs, choose the one with higher rank
    - Order is important in the specification
    - Put keywords first!
C scanner

Declaring
\% include "parser.tab.h"
\%

Identifier
{([a-zA-Z][0-9a-zA-Z_]*)}
any_white
{[ \011\013\014\015]}\%

Octal_escape
{[0-7][^'"\n]*}
\%

For
{for} { lval.tok = get_pos(); return ctokFOR;}

If
{if} { lval.tok = get_pos(); return ctokIF;}

Identifier
{[identifier]} { lval.tok = get_pos();
if (is_typename(cbtext)) return TYPEDEFname;
else return IDENTIFIER; }

Decimal_constant
{[decimal_constant]} { lval.exprN = atoi(cbtext);
return INTEGERconstant; }

\%
...any special code...

Implementation

- Table driven
  - Read and classify character
  - Select action
  - Find the next state, assign to state variable
  - Repeat

- Alternative: direct coding
  - Each state is a chunk of code
  - Transitions test and branch directly
  - Very ugly code – but who cares?
  - Very efficient

Building a lexer

Specification
"if"
"while"
[a-zA-Z] [a-zA-Z0-9]*
[0-9] [0-9]*

NFA for each RE

Giant NFA

Giant DFA

Table or code

Building scanners

- The point
  - Theory lets us automate construction
  - Language designer writes down regular expressions
  - Generator does: RE \( \rightarrow \) NFA \( \rightarrow \) DFA \( \rightarrow \) code
  - Reliably produces fast, robust scanners

- Works for most modern languages
  Think twice about language features that defeat the DFA-based scanners

Next time...

- Grammars and parsing
  - Project 1.5:
    - Generate a DFA for a regular expression
    - I’ll give you classes for REs (essentially, an IR for regular expressions)
    - You will apply the three steps: Thompson’s construction, subset construction, DFA minimization
    - Output: labeled graphs for “dot”