Scanning
(Lexical Analysis)

The Front End

- The purpose of the front end is to deal with the input language
  - Perform a membership test: code a source language?
  - Is the program well-formed (semantically)?
  - Build an IR version of the code for the rest of the compiler

The front end is not monolithic

The Front End

- Maps stream of characters into words
  - Basic unit of syntax
    - \( x \times y \): become
    - \(<id, x>, <eq>, <id, x>, <pl>, <id, y>, <eq, x>\)
- Characters that form a word are its lexeme
- Its part of speech (or syntactic category) is called its token type
- Scanner discards white space & (often) comments

The Front End

- Checks stream of classified words (parts of speech) for grammatical correctness
- Determines if code is syntactically well-formed
- Guides checking at deeper levels than syntax
- Builds an IR representation of the code

We'll get to parsing in the next lectures

The Big Picture

- Language syntax is specified with parts of speech, not words
- Syntax checking matches parts of speech against a grammar

1. goal -> expr
2. expr -> expr op term
3. | term
4. term -> number
5. \(|\)
6. op -> +
7. \(|\)

\( S = \text{goal} \)
\( T = \{\text{number, id, \ast, -}\} \)
\( N = \{\text{goal, expr, term, op}\} \)
\( F = \{1, 2, 3, 4, 5, 6, 7\} \)

Why study lexical analysis?

- We want to avoid writing scanners by hand

Goals:
  - To simplify specification & implementation of scanners
  - To understand the underlying techniques and technologies

Represent words as indices into a global table

Specifications written as "regular expressions"
Regular Expressions

Lexical patterns form a regular language

 Regular expressions (REs) describe regular languages

Regular Expression (over alphabet \( \Sigma \))
- \( \varepsilon \) is a RE denoting the set \( \{ \varepsilon \} \)
- If \( a \) is in \( \Sigma \), then \( a \) is a RE denoting \( \{ a \} \)
- If \( x \) and \( y \) are REs denoting \( L(x) \) and \( L(y) \) then
  - \( x \cdot y \) is an RE denoting \( L(x) \cdot L(y) \)
  - \( x^* \) is an RE denoting \( L(x)^* \)

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 = { x</td>
</tr>
<tr>
<td>Concatenation of ( L ) and ( M )</td>
<td>( L \cdot M = { x \cdot y</td>
</tr>
<tr>
<td>Kleene closure of ( L )</td>
<td>( L^* = \cup_{0 \leq n} L^n )</td>
</tr>
<tr>
<td>Positive closure of ( L )</td>
<td>( L^+ = \cup_{1 \leq n} L^n )</td>
</tr>
</tbody>
</table>

These definitions should be well known.

Examples of Regular Expressions

Identifiers:
- Letter \( \rightarrow \{ a | b | c | \ldots | z \} \)
- Digit \( \rightarrow \{ 0 | 1 | 2 | \ldots | 9 \} \)
- Identifier \( \rightarrow \) Letter, Letter | Digit |

Numbers:
- Integer \( \rightarrow \{ 0 | 1 | 2 | \ldots | 9 \} \)
- Decimal \( \rightarrow \) Integer | Digit |
- Real \( \rightarrow \) Integer | Decimal | Integer | Digit |
- Complex \( \rightarrow \) Real | Real |

Regular Expressions

Regular expressions can be used to specify the words to be translated to parts of speech by a lexical analyzer.

Using results from automata theory and theory of algorithms, we can automatically build recognizers from regular expressions.

We study REs and associated theory to automate scanner construction!

Example

Consider the problem of recognizing ILOC register names
- Register \( \rightarrow \{ a | b | c | \ldots | z \} \)
- Allows registers of arbitrary number
- Requires at least one digit

RE corresponds to a recognizer (or DFA)

DFA operation
- Start in state \( S_0 \)
- DFA accepts a word \( s \) if \( s \) leaves it in a final state (\( S_2 \))

<table>
<thead>
<tr>
<th>( S_0 )</th>
<th>( a )</th>
<th>( b )</th>
<th>( c )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_0 )</td>
<td>( S_1 )</td>
<td>( S_2 )</td>
<td>( S_2 )</td>
</tr>
</tbody>
</table>

Transitions on other inputs go to an error state, \( S_e \).
Example (continued)

To be useful, recognizer must turn into code

| Char ← next character | State ← s₀ |
| while (Char = EOF) |
| State ← δ(State,Char) |
| Char ← next character |
| if (State is a final state) then report success else report failure |

Skeleton recognizer

Table encoding RE

Example (continued)

To be useful, recognizer must turn into code

| Char ← next character | State ← s₀ |
| while (Char = EOF) |
| State ← δ(State,Char) |
| Char ← next character |
| if (State is a final state) then report success else report failure |

Skeleton recognizer

Table encoding RE

What if we need a tighter specification?

- Digit Digit* allows arbitrary numbers
  - Accepts 000000
  - Accepts 999999
- What if we want to limit it to 0 through 31?

Write a tighter regular expression

\[ \rightarrow \text{Register} \rightarrow c \ ( \{0,1,2,3\} \ (0|1|2|3|4|5|6|7|8|9) \ (3|30|31) ) \]


Tighter register specification (continued)

The DFA for

\[ \rightarrow \text{Register} \rightarrow c \ ( \{0,1,2,3\} \ (0|1|2|3|4|5|6|7|8|9) \ (3|30|31) ) \]

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

Tighter register specification (continued)

<table>
<thead>
<tr>
<th>δ</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>s₀</td>
<td>s₁</td>
<td>s₂</td>
<td>s₂</td>
<td>s₂</td>
<td>s₂</td>
<td>s₂</td>
</tr>
<tr>
<td>s₁</td>
<td>s₂</td>
<td>s₁</td>
<td>s₁</td>
<td>s₁</td>
<td>s₁</td>
<td>s₁</td>
</tr>
<tr>
<td>s₂</td>
<td>s₁</td>
<td>s₁</td>
<td>s₁</td>
<td>s₁</td>
<td>s₁</td>
<td>s₁</td>
</tr>
<tr>
<td>s₃</td>
<td>s₂</td>
<td>s₂</td>
<td>s₂</td>
<td>s₂</td>
<td>s₂</td>
<td>s₂</td>
</tr>
<tr>
<td>s₄</td>
<td>s₁</td>
<td>s₁</td>
<td>s₁</td>
<td>s₁</td>
<td>s₁</td>
<td>s₁</td>
</tr>
<tr>
<td>s₅</td>
<td>s₁</td>
<td>s₁</td>
<td>s₁</td>
<td>s₁</td>
<td>s₁</td>
<td>s₁</td>
</tr>
<tr>
<td>s₆</td>
<td>s₁</td>
<td>s₁</td>
<td>s₁</td>
<td>s₁</td>
<td>s₁</td>
<td>s₁</td>
</tr>
<tr>
<td>s₇</td>
<td>s₁</td>
<td>s₁</td>
<td>s₁</td>
<td>s₁</td>
<td>s₁</td>
<td>s₁</td>
</tr>
</tbody>
</table>

Table encoding RE for the tighter register specification

Constructing a Scanner - Quick Review

- The scanner is the first stage in the front end
- Specifications can be expressed using regular expressions
- Build tables and code from a DFA
Goal

- We will show how to construct a finite state automaton to recognize any RE
- Overview:
  - Direct construction of a nondeterministic finite automaton (NFA) to recognize a given RE
  - Requires \( \varepsilon \)-transitions to combine regular subexpressions
  - Construct a deterministic finite automaton (DFA) to simulate the NFA
  - Use a set-of-states construction
  - Minimize the number of states
  - Hopcroft state minimization algorithm
  - Generate the scanner code
  - Additional specifications needed for details

More Regular Expressions

- All strings of 1s and 0s ending in a 1
  \[(11)^*\]
- All strings over lowercase letters where the vowels (a,e,i,o, & u) occur exactly once, in ascending order
  \[\text{Cons} \rightarrow \text{bcldfghijklmlnoprstvwxyz}\]
- All strings of 1s and 0s that do not contain three Os in a row:
  \[\{ (1\,|\,0)^*\, (0\,|\,1\,|\,0\,|\,1)^* \}\]

Non-deterministic Finite Automata

- Each RE corresponds to a deterministic finite automaton (DFA)
- May be hard to directly construct the right DFA

What about an RE such as \((ab)^+\)?

\[S_1 \rightarrow S_2, S_2 \rightarrow \varepsilon\]

This is a little different
- \(S_2\) has a transition on \(\varepsilon\)
- \(S_2\) has two transitions on \(a\)
- This is a non-deterministic finite automaton (NFA)

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 → Obviously
- 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
Automating 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, Flex, JLex work along these lines.
- Algorithms are well-known and well-understood.
- Key issue is interface to parser (define all parts of speech).
- You could build one in a weekend.

RE → NFA (Thompson’s construction)

Key idea:
- NFA pattern for each symbol and each operator.
- Each NFA has a single start and accept state.
- Join them with ε moves in precedence order.

Example of Thompson’s Construction (cont’)

4. \( (\text{b} | \varepsilon)’ \)

Of course, a human would design something simpler...

But, we can automate production of the more complex one...

Automating Scanner Construction

RE → NFA (Thompson’s construction)
- Build an NFA for each term.
- Combine them with ε-moves.

NFA → DFA (subset construction)
- Build the simulation.

DFA → Minimal DFA
- Hopcroft’s algorithm.

DFA → RE (Not part of the scanner construction)
- All pairs, all paths problem.
- Take the union of all paths from \( q_0 \) to an accepting state.

Example of Thompson’s Construction

Let’s try \( \textbf{g} (\text{b} | \varepsilon)’ \)

1. \( \varepsilon \), b, \( \delta \),

2. \( b | \varepsilon \)

3. \( b | \varepsilon ’ \)

NFA → DFA with Subset Construction

Need to build a simulation of the NFA.

Two key functions
- \( \text{Move}(s, a) \) is set of states reachable from \( s \) by \( a \).
- \( \varepsilon\text{-closure}(s) \) is set of states reachable from \( s \) by \( \varepsilon \).

The algorithm:
- Start state derived from \( s_0 \) of the NFA.
- Take its \( \varepsilon\text{-closure} \).
- Take the image of \( S_0 \), \( \text{move}(S_0, a) \) for each \( a \in \Sigma \), and take
its \( \varepsilon\text{-closure} \).
- Iterate until no more states are added.

Sounds more complex than it is.

Ken Thompson, CACM, 1968
NFA → DFA with Subset Construction

The algorithm:

\[ s_0 \leftarrow \text{closure}(q_0) \]
\[ \text{add } s_0 \text{ to } S \]
while (S is still changing) for each \( s_i \in S \)
\[ s_i \leftarrow \text{closure}(\text{move}(s_i)) \]
if \( \{ s_i, \Sigma \} \) then add \( s_i \) to \( S \) as \( s_i \)
\[ T[\{s_i, \Sigma\}] = s_j \]

Let’s think about why this works

The algorithm halts:
1. \( S \) contains no duplicates (test before adding)
2. \( 2^\Sigma \) is finite
3. While loop adds to \( S \), but does not remove from \( S \) (monotone)
4. The loop halts

\( S \) contains all the reachable NFA states
If tries each character in each \( s_i \), it builds every possible NFA configuration.
\( S \) and \( T \) form the DFA

Example of a fixed-point computation
- Monotone construction of some finite set
- Halts when it stops adding to the set
- Proofs of halting & correctness are similar
- These computations arise in many contexts

Other fixed-point computations
- Canonical construction of sets of LR(1) items
  - Quite similar to the subset construction
- Classic data flow analysis
  - Solving sets of simultaneous set equations

We will see many more fixed-point computations

NFA → DFA with Subset Construction

\[ a \cdot b \cdot e^* \]

Applying the subset construction:

<table>
<thead>
<tr>
<th>NFA states</th>
<th>( a )</th>
<th>( b )</th>
<th>( e )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_0 )</td>
<td>( s_0 )</td>
<td>( s_0 )</td>
<td>( s_0 )</td>
</tr>
<tr>
<td>( s_0 )</td>
<td>( s_0 \cdot s_1, s_2 )</td>
<td>( s_0 \cdot s_1, s_2 )</td>
<td>( s_0 \cdot s_1, s_2 )</td>
</tr>
<tr>
<td>( s_1 )</td>
<td>( s_1 \cdot s_0, s_1 )</td>
<td>( s_1 \cdot s_0, s_1 )</td>
<td>( s_1 \cdot s_0, s_1 )</td>
</tr>
<tr>
<td>( s_1 )</td>
<td>( s_1 \cdot s_0, s_1 )</td>
<td>( s_1 \cdot s_0, s_1 )</td>
<td>( s_1 \cdot s_0, s_1 )</td>
</tr>
</tbody>
</table>

Final states

\[ s_0 \rightarrow s_3 \rightarrow s_2 \rightarrow s_1 \]

Automating Scanner Construction

RE → NFA (Thompson’s construction)
- Build an NFA for each term
- Combine them with \( \epsilon \)-moves

NFA → DFA (subset construction)
- Build the simulation

DFA → Minimal DFA
- Hopcroft’s algorithm

DFA → RE (not really part of scanner construction)
- All pairs, all paths problem
- Union together paths from \( s_i \) to a final state

DFA Minimization

The Big Picture
- Discover sets of equivalent states
- Represent each such set with just one state
DFA Minimization

The Big Picture

- Discover sets of equivalent states
- Represent each such set with just one state

Two states are equivalent if and only if:

- $\forall a \in \Sigma$, transitions on $a$ lead to equivalent states (DFA)
- $a$-transitions to distinct sets $\Rightarrow$ states must be in distinct sets

**Details of the algorithm**

- Group states into maximal-size sets, optimistically
- Iteratively subdivide those sets, as needed
- States that remain grouped together are equivalent

Initial partition, $P_0$, has two sets: $(F)$ & $(Q-F)$ $(D = (Q, \Sigma, \delta, F, F))$

Splitting a set ("partitioning a set by $g"$)

- Assume $q_s \not\sim q_g$, $s$, and $\delta(q_s, a) = q_s$, $\delta(q_g, a) = q_g$
- If $q_s$ & $q_g$ are not in the same set, then $s$ must be split
- $q_s$ has transition on $a$, $q_s$ does not $\Rightarrow$ $g$ splits $s$

**Abbreviated Register Specification**

Start with a regular expression

$r0 \mid r1 \mid r2 \mid r3 \mid r4 \mid r5 \mid r6 \mid r7 \mid r8 \mid r9$

Note: "$|$" is left associative

DFA Minimization

Why does this work?

- Partition $P = 2^Q$
- Start off with 2 subsets of $Q$
- While loop takes $P_0 \rightarrow P$, by splitting 1 or more sets
- $P_i$ is at least one step closer to the partition with $|Q|$ sets
- Maximum of $|Q|$ splits

Note that:

- Partitions are never combined

DFA Minimization

Then apply the minimization algorithm

<table>
<thead>
<tr>
<th>Current Partition</th>
<th>Split on</th>
<th>Final states</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_0$</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>$Q_0 \equiv { q_s }$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To produce the minimal DFA

We observed that a human would design a simpler automaton than Thompson's construction & the subset construction did.

Minimizing that DFA produces the one that a human would design!
**Abbreviated Register Specification**

Thompson's construction produces

```
                  r
                /   \
              0,1,2,3,4
            /     \       \
         5,6,7,8,9    r
```

**Abbreviated Register Specification**

The DFA minimization algorithm builds

```
0,1,2,3,4, 
5,6,7,8,9
```

This looks like what a skilled compiler writer would do!

**Abbreviated Register Specification**

The subset construction builds

This is a DFA, but it has a lot of states...

**Abbreviated Register Specification**

The Cycle of Constructions

---

**Limits of Regular Languages**

Advantages of Regular Expressions
- Simple & powerful notation for specifying patterns
- Automatic construction of fast recognizers
- Many kinds of syntax can be specified with REs

Example — an expression grammar
```
Term → [0-9A-Z]<0-9A-Z-1|2>
Op → + | - | * | / |
Expr → (Term Op)* Term
```

Of course, this would generate a DFA...

If REs are so useful...

*Why not use them for everything?*

**What can be so hard?**

Poor language design can complicate scanning
- Reserved words are important
- Significant blanks
- String constants with special characters
- Limited identifier "length"

---

CMSC 430 Lecture 2 46

CMSC 430 Lecture 2 47

CMSC 430 Lecture 2 48