COMP 181

Lecture 4

More scanning

September 14, 2006

Programming assignment

- Big picture
  - Develop a compiler for *Decaf*, a subset of Java
  - 4 parts – we'll skip the lexer
    - PA1: an interpreter
    - PA2: parser
    - PA3: light version of semantic checker/code generator
    - PA4: full version of code generator
- Today: start PA1

PA1: Interpreter

- *SkimDecaf*: subset of subset of Java
  - One class, one method
  - No variable declarations
  - Only variables of type int
  - Only 4 operators: +, -, *, /
  - Two statements: if and while
- Your Job:
  - Interpreter
  - Pretty printer

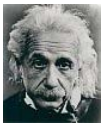
Starter kit

- What I'll give you
  - Code skeleton
  - Parser: produces AST for *SkimDecaf* programs
- What you do:
  - Learn about the AST structure
  - Fill in code skeleton
  - Develop some test cases

Environment

- JDK
  - Need access to java and javac
  - I have installed jdk 1.5 for x86/Linux:
    `/g/181/2006f/jdk1.5.0_08`
- Eclipse
  - This assignment relies on the Eclipse parser
  - I have installed Eclipse for x86/Linux:
    `/g/181/2006f/eclipse`
- Demo…

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…

Scanner construction

[0] Define tokens as regular expressions

[1] Construct NFA for all REs
- Connect REs with ε transitions
- ***Thompson's construction***

[2] Convert NFA into a DFA
- DFA is a simulation of NFA
- Possible much larger than NFA

[3] Minimize the DFA
- ***Hopcroft's algorithm***

[4] Generate implementation

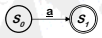
[1] Thompson's construction

- ***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***

Thompson's construction

- Given:

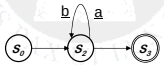
- Build **xy**

- Why can't we do this?

Need for ε transitions

- What if **x** and **y** look like this:

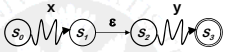
- Then **xy** ends up like this:



Thompson's construction

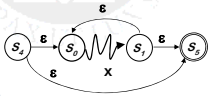
- Given:

- **xy**



- **x|y**

- **x***

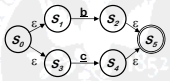


Example

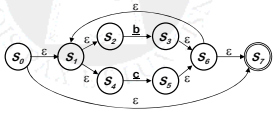
Regular expression: a (b | c)*

- a, b, & c

- b | c

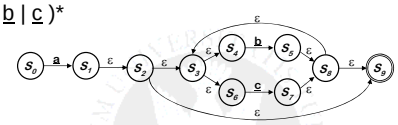


- (b | c)*

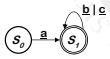


Example

- a (b | c)*



- Note: a human could design something simpler…

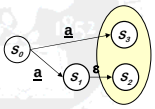


Problem

- How to implement NFA scanner code?
  - 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
    - next(s_i, a) – the set of states reachable from s_i on a
    - ε-closure(s_i) – the set of states reachable from s_i on ε

Subset construction

- Algorithm

Build a set of subs… of NFA states

```
s0 ← ε-closure(q0)
S ← { s0 }
worklist ← { s0 }
while ( worklist is not empty )
  remove s from worklist
  for each α ∈ Σ
    t ← ε-closure( next(s,α) )
    if ( t ∉ S ) then
      add t to S
      add t to worklist
      add transition (s, α, t)
```

Start with the ε-closure over the initial state

Initialize the worklist to this one subset

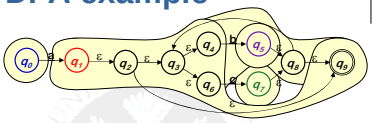
While there are more subsets on the worklist, remove the next subset

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

NFA to DFA example

a (b | c)* :



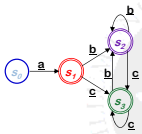
Subsets S	NFA states	a	b	c
s_0	q_0	$q_1, q_2, q_3, q_4, q_6, q_9$	none	none
s_1	$q_1, q_2, q_3, q_4, q_6, q_9$	none	$q_5, q_8, q_9, q_3, q_4, q_6$	$q_7, q_8, q_9, q_3, q_4, q_6$
s_2	$q_5, q_8, q_9, q_3, q_4, q_6$	none	(also s_2)	(also s_3)
s_3	$q_7, q_8, q_9, q_3, q_4, q_6$	none	(also s_2)	(also s_3)

ε-closure(next(s,α))

Accepting states

NFA to DFA example

- Convert each subset in S into a state:



δ	a	b	c
s_0	s_1	-	-
s_1	-	s_2	s_3
s_2	-	s_2	s_3
s_3	-	s_2	s_3

  - All transitions are deterministic
  - Use same code engine to execute
  - Smaller than NFA, but still bigger than necessary

Does it work?

- The algorithm halts
  - 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
  - ***Fixpoint*** 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
- 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

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 & $s_b \in p_i$, and $\delta(s_a, a) = s_x$, & $\delta(s_b, a) = s_y$
  - Split p_i if:
    - If s_x & s_y are not in the same set
    - If s_a has a transition on a, but s_b does not

    Intuition: one state in DFA cannot have two transitions on a

Examples

DFA minimization algorithm

```
P ← { F, {Q-F}}
while ( P is still changing)
    T ← { }
    for each set S ∈ P
        for each α ∈ Σ
            partition S by α
                into S1, and S2
            T ← T ∪ S1 ∪ S2
    if T ≠ P then
        P ← T
```

Start with two sets: final states, everything else

Build a new partitioning

For each set and each input symbol, try to partition the set

Collect the resulting sets in a new partition, see if it's different

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 $P_i \rightarrow P_{i+1}$ 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

DFA minimization

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

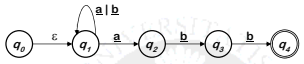
⇨ This version looks at each $S \in P$ many fewer times
Well-known, widely used algorithm due to John Hopcroft

Hopcroft's algorithm

```
W ← {F, Q-F}; P ← {F, Q-F}; // W is the worklist, P the current partition
while ( W is not empty ) do begin
    select and remove S from W; // S is a set of states
    for each α in Σ do begin
        let I_α ← δ_α^-1( S ); // I_α is set of all states that can reach S on α
        for each R in P such that R ∩ I_α is not empty
            and R is not contained in I_α do begin
            partition R into R1 and R2 such that R1 ← R ∩ I_α; R2 ← R – R1;
            replace R in P with R1 and R2;
            if R ∈ W then replace R with R1 in W and add R2 to W;
            else if || R1 || ≤ || R2 ||
                then add add R1 to W;
                else add R2 to W;
        end
    end
end
```


Full example

- Consider (a | b)* abb



- Applying the subset construction:

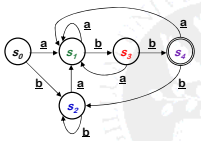
Iter.	State	Contains	ε-closure(move(s_i,a))	ε-closure(move(s_i,b))
0	s_0	q_0, q_1	q_1, q_2	q_1
1	s_1	q_1, q_2	q_1, q_2	q_1, q_3
	s_2	q_1	q_1, q_2	q_1
2	s_3	q_1, q_3	q_1, q_2	q_1, q_4
3	s_4	q_1, q_4	q_1, q_2	q_1

contains q_4 (final state)

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

Full example

- DFA for (a | b)* abb



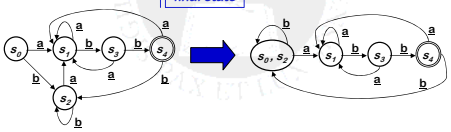
δ	a	b
s_0	s_1	s_2
s_1	s_1	s_3
s_2	s_1	s_2
s_3	s_1	s_4
s_4	s_1	s_2

- About the same size as NFA

Full example – minimization

	Current Partition	Worklist	s	Split on a	Split on b
P_0	{s_4} {s_0,s_1,s_2,s_3}	{s_4} {s_0,s_1,s_2,s_3}	{s_4}	none	{s_0, s_1, s_2} {s_3}
P_1	{s_4} {s_3} {s_0,s_1,s_2}	{s_0,s_1,s_2} {s_3}	{s_3}	none	{s_0, s_2}{s_1}
P_2	{s_4} {s_3} {s_1} {s_0,s_2}	{s_0,s_2}{s_1}	{s_1}	none	none

final state

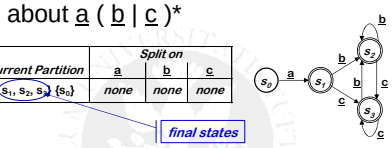


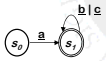
Another example

- What about a (b | c)*

	Current Partition	Split on a	Split on b	Split on c
P_0	{s_1, s_2, s_3} {s_0}	none	none	none

final states





Previously, 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!

We're done with the theory

- Take a nice, deep breath…

Application

- What do our DFAs recognize?
  Given a string, is it a token
- Problem:
  Input consists of many tokens, not just one

```
elsex = 0;
```

  - Option 1: "else", "x", "=", "0"
  - Option 2: "elsex", "=", "0"

Practical scanners

- Solution:
  - List of regular expressions
  - Longest matching RE wins
  - Ties are broken by order of Res
- How do we implement that?
  Lookahead
  - Keep reading characters as long as the DFA is satisfied
  - If no transition and we're in a final state for one of the REs:
    - Return that token
    - Go back to start state
  - If no transition and we're ***not*** in a final state → error

C scanner

Declarations

Short-hand

REs and actions

```
%{
#include "parser.tab.h"
%}

identifier      ([a-zA-Z_][0-9a-zA-Z_]*)
octal_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.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

This is how lex/flex work: states are encoded as cases in a giant switch statement

Building scanners

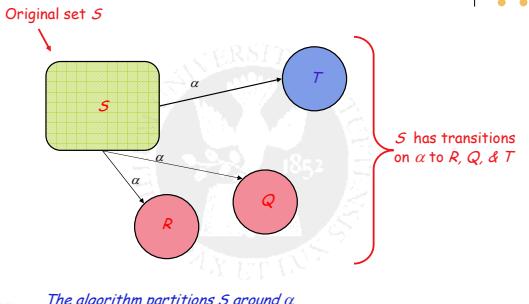
- The point
  - Theory lets us automate construction
  - Implementer writes down regular expressions
  - Generator does: RE → NFA → DFA → code
  - Reliably produces fast, robust scanners
- Works for most modern languages

Think twice about language features that defeat the DFA-based scanners

Next time…

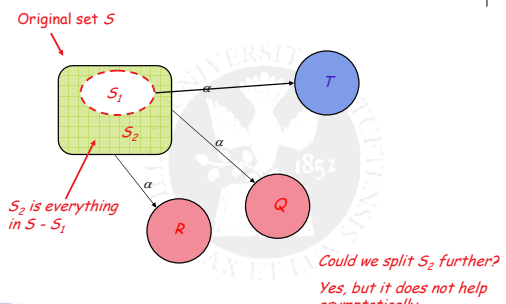
- Grammars and parsing
- Project 1:
  - SkimDecaf Intepreter
  - I'll post everything after class

Splitting S around α



The algorithm partitions S around α

Splitting S around α



Could we split S_2 further?

Yes, but it does not help asymptotically