Minimization of Deterministic Finite State Machines

We consider **deterministic** finite state machine $M = (\Sigma, Q, \delta, q_0, F)$.

Goal: build a state machine M' with the least number of states that accepts the language $L(M)$.

- we obtain a space-efficient, executable representation of a regular language

This is the process of *minimization* of M.

- an easy case of minimizing size of 'generated code' in compiler

We say that state machine M distinguishes strings w and w' iff it is not the case that ($w \in L(M)$ iff $w' \in L(M)$).

Minimization Algorithm

Step 1: Remove unreachable states

We first discard states that are not reachable from the initial state–such states are useless. In resulting machine, for each state q there exists a string s such that $\delta(q_0, s) = q$, let s_q one such string of minimal length.

(Main) Step 2: Compute Non-Equivalent States

We wish to merge states q and q' into same group as long as they "behave the same" on all future strings w, i.e.

$$\delta(q, w) \in F \text{ iff } \delta(q', w) \in F \quad (*)$$

for all w.

If the condition $(*)$ above holds, we called states **equivalent**. If the condition does **not** hold, we call states q, q' **non-equivalent**.

States q and q' are w-non-equivalent if it is not the case that ($\delta(q, w) \in F$ iff $\delta(q', w) \in F$).

Two states are non-equivalent iff they are w-non-equivalent for some string w.

Observe that

1. if $q \in F$ and $q' \notin F$ then q and q' are ϵ-non-equivalent
2. if q and q' are w-non-equivalent and we have $\delta(r, a) = q$, $\delta(r', a) = q'$ for some symbol $a \in \Sigma$, then r and r' are aw-non-equivalent
3. conversely, if r and r' are w'-non-equivalent and w is not an empty string, then for $w' = aw$ the states $\delta(r, a)$ and $\delta(r', a)$ are w-non-equivalent

These observations lead to an iterative algorithm for computing non-equivalence relation ν

1. initially put $\nu = (Q \cap F) \times (Q \setminus F)$ (only final and non-final states are initially non-equivalent)
2. repeat until no more changes: if $(r, r') \notin \nu$ and there is $a \in \Sigma$ such that $(\delta(r,a), \delta(r',a)) \in \nu$, then

$$\nu := \nu \cup \{(r, r')\}$$

Step 3: Merge States that are not non-equivalent

Relation $Q^2 \setminus \nu$ is an equivalence relation $\sim$. We define the 'factor automaton' by merging equivalent states:

- the initial state is $q_0/_\sim$
- $Q/_\sim = \{\{y \mid x \sim y\} \mid x \in Q\}$
- $F/_\sim = \{\{y \mid x \sim y\} \mid x \in F\}$
- relation $r = \{([x],[y]) \mid (x,y) \in \delta\}$ is a function, and we can use it to define a new deterministic automaton (there is a transition in the resulting automaton iff there is a transition between two states in the original automaton)

This is the minimal automaton.

Correctness of Constructed Automaton

Clearly, this algorithm terminates because in worst case all states become non-equivalent. We will prove below that the resulting value ν is the non-equivalence relation, i.e. the complement of relation given by $(*)$ above.

By induction, we can easily prove that if $(q, q') \in \nu$, then q and q' are non-equivalent. Similarly we can show that if q and q' are w-non-equivalent for w of length k, then $(q, q') \in \nu$ by step k of the algorithm. Because the algorithm terminates, this completes the proof that ν is the non-equivalence relation.

Consequently, $Q^2 \setminus \nu$ is the equivalence relation. From the definition of this equivalence it follows that if two states are equivalent, then so is the result of applying δ to them. Therefore, we have obtained a well-defined deterministic automaton.

Minimality of Constructed Automaton

Note that if two distinct states are non-equivalent, there is w such that states $\delta(q_0, s_q w)$ and $\delta(q_0, s_{q'} w)$ have different acceptance, so M distinguishes $s_q w$ and $s_{q'} w$. Now, if we take any other state machine $M' = (\Sigma, Q', \delta', q_0', F')$ with $L(M') = L(M)$, it means that $\delta'(q_0', s_q) \neq \delta'(q_0', s_{q'})$, otherwise M' would not distinguish $s_q w$ and $s_{q'} w$. So, if there are K pairwise non-equivalent states in M, then a minimal finite state machine for $L(M)$ must have at least K states. Note that if the algorithm constructs a state machine with K states, it means that $Q^2 \setminus \tau$ had K equivalence relations, which means that there exist K non-equivalent states. Therefore, any other deterministic machine will have at least K states, proving that the constructed machine is minimal.

Basic Idea of First Symbol Computation

When Exactly Does Recursive Descent Work?

When can we be sure that recursive descent parser will parse grammar correctly?

- it will accept without error exactly when string can be derived

Consider grammar without repetition construct * (eliminate it using right recursion).

Given rules

```
X ::= p
X ::= q
```

that is,

```
X ::= p | q
```

where p,q are sequences of terminals and non-terminals, we need to decide which one to use when parsing X, based on the first character of possible string given by p and q.

- first(p) - first characters of strings that p can generate
- first(q) - first characters of strings that q can generate
- requirement: first(p) and first(q) are **disjoint**

How to choose alternative: check whether current token belongs to first(p) or first(q)

Computing 'first' in Simple Case

Assume for now

- no non-terminal derives empty string, that is:

For every terminal X, if X =⇒* w and w is a string of terminals, then w is non-empty

We then have

- first(X ...) = first(X)
- first(”a” ...) = {a}

We compute first(p) set of terminals for

- every right-hand side alternative p, and
- every non-terminal X

Example grammar:

```
S ::= X | Y
X ::= "b" | S Y
Y ::= "a" X "b" | Y "b"
```

Equations:

- first(S) = first(X|Y) = first(X) ∪ first(Y)
- first(X) = first(”b” | S Y) = first(”b”) ∪ first(S Y) = {b} ∪ first(S)
- first(Y) = first(”a” X “b”|Y “b”) = first(”a” X “b”) ∪ first(Y “b”) = {a} ∪ first(Y)

How to solve equations for first?

Expansion: first(S) = first(X) ∪ first(Y) = {b} ∪ first(S) ∪ {a} ∪ first(Y)

- could keep expanding forever
- does further expansion make difference?
- is there a solution?
- is there unique solution?

Bottom up computation, while there is change:

- initially all sets are empty
- if right hand side is bigger, add different to left-hand side

Solving equations

- first(S) = first(X) ∪ first(Y)
- first(X) = {b} ∪ first(S)
- first(Y) = {a} ∪ first(Y)

bottom up

first(S)	first(X)	first(Y)
{}	{}	{}
{}	{b}	{a}
{a,b}	{b}	{a}
{a,b}	{a,b}	{a}
{a,b}	{a,b}	{a}

Does this process terminate?

- all sets are increasing
- a finite number of symbols in grammar

There is a unique **least** solution

- this is what we want to compute
- the above bottom up algorithm computes it

General Remark:

- this is an example of a ‘fixed point’ computation algorithm
- also be useful for semantic analysis, later

Nullable Non-terminals

In general, a non-terminal can expand to empty string

- example: statement sequence in while language grammar

first(Y Z) = first(Y)? what if Y can derive empty string?

A **sequence** of non-terminals is **nullable** if it can derive an empty string

- this is case iff each non-terminal is **nullable**

Computing nullable non-terminals:

- empty string is nullable
- if one right-hand side of non-terminal is nullable, so is the non-terminal

Algorithm:

```
nullable = {}
changed = true
while (changed) {
  changed = false
  for each non-terminal X
  if X is not nullable and either
  1) grammar contains rule
    X ::= "" | ...
  or
  2) grammar contains rule
    X ::= Y1 ... Yn | ...
      and
    {Y1,...,Yn} is contained in nullable
  then
    nullable = nullable union {X}
    changed = true
}
```

Computing First Given Nullable

Computing first(X), given rule X = Y_1 ... Y_i ... Y_k

- if Y_1,..., Y_{i-1} are all nullable, then add first(Y_i) to first(X)

Then repeat until no change, as before.

Computing Follow Sets

The Need for Follow

What if we have

```
X = Y Z | U
```

and U is nullable? When can we choose a nullable alternative (U)?

- if current token is either in first(U) or it could **follow** non-terminal X

t is in follow(X), if there exists a derivation containing substring X t

Example of language with ‘named blocks’:

```
statements ::= "" | statement statements
statement ::= assign | block
assign ::= ID "=" (ID|INT) ";"
block ::= "beginof" ID statements ID "ends"
```

Try to parse

```
beginof myPrettyCode
  x = 3;
  y = x;
myPrettyCode ends
```

Problem parsing ‘statements’:

- identifier could start alternative ‘statement statements’
- identifier could follow ‘statements’, so we may wish to parse “”

Computing follow(Y_i), given rule X = Y_1 ... Y_i ... Y_j ... Y_k

- add first(Y_j), if Y_{i+1},..., Y_{j-1} are all nullable
- add follow(X), if Y_{i+1}, ..., Y_k are all nullable

Possible computation order:

- nullable
- first
- follow

Example: compute these values for grammar above

```
follow = {}
first
  statements {ID, "beginof"}
  statement  {ID, "beginof"}
  assign     {ID}
  block      {"beginof"}
follow
  statements {ID}
```

```
  statement  {ID, "beginof"}
  assign     {ID, "beginof"}
  block      {ID, "beginof"}
```

The grammar cannot be parsed because we have

```
statements ::= "" | statement statements
```

where

- statements $\in$ nullable
- first(statements) $\cap$ follow(statements) = {ID} $\neq \emptyset$

If the parser sees ID, it does not know if it should

- finish parsing ‘statements’ or
- parse another ‘statement’

Algorithm for Computing First and Follow Sets

```
nullable = {}
foreach nonterminal X:
  first(X)={}
  follow(X)={}
for each terminal Y:
  first(Y)={Y}

repeat
  foreach grammar rule X ::= Y(1) ... Y(k)
  if k=0 or {Y(1),...,Y(k)} subset of nullable then
    nullable = nullable union {X}
  for i = 1 to k
    for j = i+1 to k
      if i=1 or {Y(1),...,Y(i-1)} subset of nullable then
        first(X) = first(X) union first(Y(i))
      if i=k or {Y(i+1),...Y(k)} subset of nullable then
        follow(Y(i)) = follow(Y(i)) union follow(X)
      if i+1=j or {Y(i+1),...,Y(j-1)} subset of nullable then
        follow(Y(i)) = follow(Y(i)) union first(Y(j))
until none of nullable,first,follow changed in last iteration
```

Table-Driven Parser for Balanced Parentheses

First Grammar

It has three alternatives:

```
S ::= "" | ( S ) | S S
      1.     2.     3.
```

Goal is to figure out which alternative to use when – convert | into if-then-else.

Compute:

```
nullable = { S }
first(S) = { ( }
follow(S) = { ( , ) }
```

Parse Table:

	(	**)**
S	1,2,3	

Because we have duplicate entries, we cannot use this to build a parser.

Second Grammar

```
S ::= "" | F S
F ::= ( S )
```

Again compute:

```
nullable = { S }
first(S) = { ( }
follow(S) = { ) }
```

	(	**)**
S	2	1
F	1	{}

Recursive Descent Parser

Constructed mechanically:

```
def S = {
  if (lexer.token==OpenP) { F; S }
  else if (lexer.token==ClosedP) ()
  else error("Expected '(' or ')'")
}
def F = {
  if (lexer.token==OpenP) {
    lexer.next
    S
```

```
    skip(ClosedP)
  } else error("Expected '('")
}
```

Simplified:

```
def S = if (lexer.token==OpenP) { F; S }
def F = { skip(OpenP); S; skip(ClosedP) }
```

Top-Down Parser Using a Stack

```
stack push "S"
while (!stack empty) {
  val X = stack pop
  if (X isTerminal) skip(X)
  else {
    if (X=="S") {
      if (lexer.token==OpenP) stack push "S" push "F"
    } else if (X=="F") stack push ClosedP push "S" push OpenP
  }
}
```

LL(1) Table-Driven Parsing Overview

First, compute nullable, first, follow

Then, make **parsing table** which stores the alternative, given

- non-terminal being parsed (in which procedure we are)
- current token

Given (X ::= p_1 | ... | p_n) we insert alternative j into table iff

- t $\in$ first(p_j), or
- nullable(p_j) and t $\in$ follow(X)

If in parsing table we have two or more alternatives for same token and non-terminal:

- we have **conflict**
- we cannot parse grammar using recursive descent

Otherwise, we say that the grammar is **LL(1)**

- **L**eft-to-right parse (the way input is examined)
- **L**eftmost derivation (expand leftmost non-terminal first–recursion in descent does this)
- **(1)** token lookahead (current token)

What about empty entries?

- they indicate errors
- report that we expect one of tokens in
    - first(X), if X $\notin$ nullable
    - first(X) $\cup$ follow(X), if X $\in$ nullable

Building LL Parsing Table

Parsing table for LL parser is of form

```
choice : Nonterminal -> Token -> Set[Int]
```

We computing it for each nonterminal X by looking at all right-hand sides of X. Denote i-th right-hand side of X by p(X,i):

```
X ::= p(X,1) | ... | p(X,i) | ... | p(X,n)
```

Compute choice(X)(t) as follows:

```
choice(X)(t) = {i | t in first(p(X,i)) or (p(X,i) nullable and t in follow(X)}
```

(Z(1)...Z(k) is nullable if all Z(1),...,Z(k) are nullable non-terminals.)

Size of choice(X)(t):

- we require that each entry choice(X)(t) has at most one element (otherwise not LL(1))
- empty entries are normal, if encountered they indicate syntax error in input

Interpreting LL Parsing Table

This is the top-down parser:

```
var stack : Stack[GrammarSymbol]
stack.push(EOF);
stack.push(StartNonterminal);
lex = new Lexer(inputFile)
while (true) {
* X = stack.pop
  t = lex.curent
  if isTerminal(X)
    if (t==X)
      if (X==EOF) return success
      else lex.next // eat token t
    else
      parseError("Expected " + X)
  else // non-terminal
    cs = choice(X)(t)
    cs match {
    case {i} => // exactly one choice
      rhs = p(X,i) // choose correct right-hand side
*     stack.push(reverse(rhs))
    case {} => parseError("Parser expected an element of " + unionOfAll(choice(X)))
    case _ => crash("wrong parse table, not LL(1)")
    }
}
```

The lines marked with * give us the essence of this parser: it pops non-terminals from stack and replaces them with the right-hand side of a production rule.

When we write recursive descent procedures by hand, the stack is implicit in the use of recursive procedures.

Note: the program above corresponds to a deterministic push-down automaton that parses the LL(1) grammar

- non-deterministic push down automata correspond to all grammars
- determinization of push down automata in general is not possible, non-deterministic ones are more expressive

Compilation as Tree Transformation

Motivation:

- elegant and efficient compilation for conditionals
- compilation for more complex control structures

To describe this compilation we introduce an imaginary, big, instruction

```
branch(c,nThen,nElse)
```

Here

- c is a potentially complex Java boolean expression
- nThen is label to jump to when c evaluates to true
- nFalse is label to jump to when c evaluates to false

Next, we show how to expand branch(c,nThen,nElse) into actual instructions. This is a recursive process.

Using 'branch' in Compilation

```
[[ if (c) sThen else sElse ]] =
        branch(c,nThen,nElse)
nThen:  [[ sThen ]]
        goto nAfter
nElse:  [[ sElse ]]
nAfter:
```

```
[[ while (c) s ]] =
```

```
lBegin: branch(c,start,lExit)
start:  [[ s ]]
        goto lBegin
lExit:
```

Decomposing Condition in 'branch'

Negation

```
branch(!c,nThen,nElse) =
         branch(c,nElse,nThen)
```

And

```
branch(c1 && c2,nThen,nElse) =
         branch(c1,nNext,nElse)
  nNext: branch(c2,nThen,nElse)
```

Here, nNext is a fresh label.

Or

```
branch(c1 || c2,nThen,nElse) =

           branch(c1,nThen,nNext)
    nNext: branch(c2,nThen,nElse)
```

Boolean Constant

```
branch(true,nThen,nElse) =
        goto nThen

branch(false,nThen,nElse) =
        goto nElse
```

Boolean Variable

Option one:

```
branch(xN,nThen,nElse) =
  iload_N
  ifeq nElse
  goto nThen
```

Option two:

```
branch(xN,nThen,nElse) =
  iload_N
  ifne nThen
  goto nElse
```

Relation

Option one:

```
branch(e1 R e2,nThen,nElse) =
         [[ e1 ]]
         [[ e2 ]]
         if_cmpR nThen
         goto nElse
```

Option two:

```
branch(e1 R e2,nThen,nElse) =
         [[ e1 ]]
         [[ e2 ]]
         if_cmpNegR nElse
         goto nThen
```

Storing Result into Boolean Variable

What if we need to compute

```
x = c
```

where x,c are boolean?

- What are nThen,nElse labels?

Producing boolean expression on stack:

```
[[ c ]] =
        branch(c,nThen,nElse)
nThen:  iconst_1
        goto nAfter
nElse:  iconst_0
nAfter:
```

Then we can store the value as usual

Simple Peephole Optimizations

Note also that we can eliminate the pattern

```
   goto L
L:
```

if it is generated in the process above

We can pick the option that eliminates the jump

We can detect this kind of optimization by looking only at neighboring instructions

- example of ‘peephole optimization’

```
Generated instructions:
-----------------------------------
|   |   |   |   |   |   |   |   |
-----------------------------------
    |               |
    ----------------
            |
       optimizer looks at a 'window' of instructions
```

Other examples:

- recognize pattern for ‘x=x+c’, replace with iinc
- recognize copying of boolean variables (b1=b2) - no need for ‘branch’
- how to compile this assignment: b1= b2 && b3; (no better than simple scheme, but for larger expressions and simple comparisons we have an improvement)

Further advanced reading

- Compiling with Continuations
- The essence of compiling with continuations