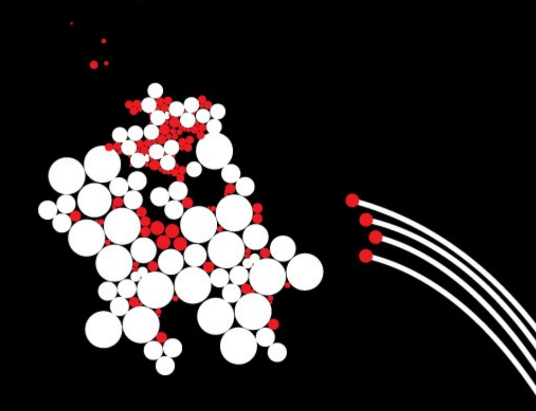
UNIVERSITY OF TWENTE.

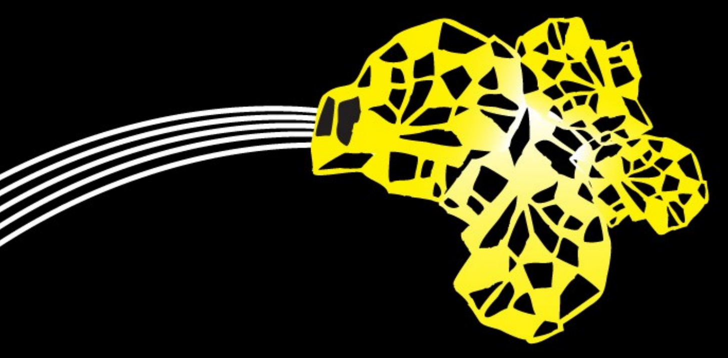


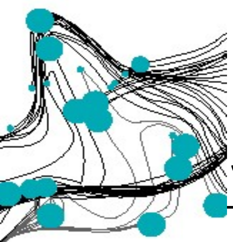
COMPILER CONSTRUCTION:

RECOGNIZING WORDS

MODULE 8: PROGRAMMING PARADIGMS

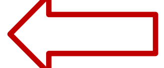
25 APRIL 2018





SEEN LAST TIME

- Structure of compiler
- Front end
  - Scanning
  - Parsing
  - Type checking
- Optimizer
- Back end



To be continued today

Scanner: turns a sequence of characters into a sequence of tokens of language-specific token types

DETERMINISTIC FINITE AUTOMATA (DFA'S)

Recognizing the keyword token "new" (pseudocode):

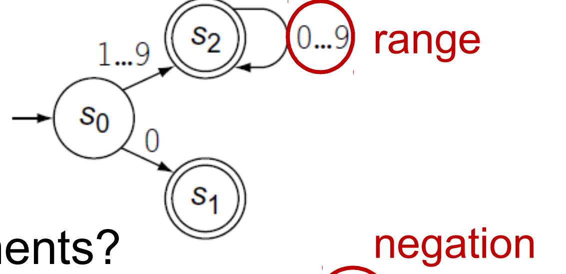
```
c ← NextChar();
if (c = 'n')
   then begin;
      c ← NextChar();
      if (c = 'e')
         then begin;
            c ← NextChar();
            if (c = 'w')
               then report success;
               else try something else;
         end;
         else try something else;
   end;
   else try something else;
```

This requires *precisely* the information of the following DFA:

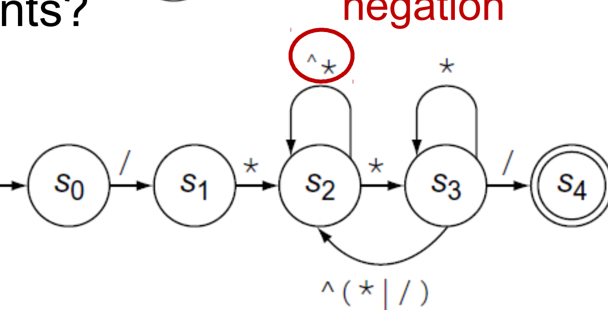
s_0 --n--> s_1 --e--> s_2 --w--> s_3

MORE COMPLEX TOKEN TYPES

- Arbitrary numbers (non-empty sequences of digits, not 00, 000 etc)?



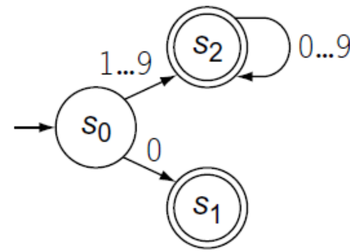
- Java block comments?



- Note the following DFA elements:
  - Combined labels on single transition (0..9 etc)
  - Loops (giving rise to arbitrary-length tokens)
  - Negation and choice (^* and ^(* | /))
  - Multiple accepting states in one DFA

GENERAL ACCEPTANCE ALGORITHM (SINGLE TOKEN)

- From a DFA



- Create a transition table
  - = transition fun
  - = error state

δ	0	1	2	3	4	5	6	7	8	9	**Other**
s_0	s_1	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_e
s_1	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e
s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_e
s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e

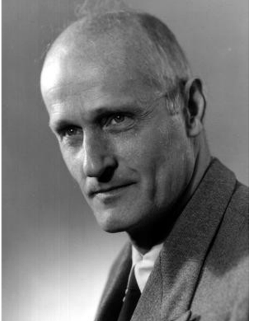
- Run the algorithm
  - *NextChar()* reads from input
  - *eof* stands for "end of file"

```
char ← NextChar( );
state ← s0 ;

while (char ≠ eof and state ≠ se) do
    state ← δ(state,char);
    char ← NextChar( );
end;

if (state ∈ SA)
    then report acceptance;
    else report failure;
```

Set of accepting states

REGULAR EXPRESSIONS (RE'S)

- RE's allow equivalent, more concise description of token types
- RE's are built from
  - Single characters, e.g., a or 0 or *
  - Choice (*alternation*), e.g., a | b or 0 | A | a. Shorthand: ranges 0..9
  - Sequences (*concatenation*), e.g., (a..z) (0..9 | a..z)
  - Repetitions (*closure/Kleene star*), e.g., (a..z | 0..9)*
  - Auxiliaries: ^(a..z) and (a..z)+ and (a..z)?
- Arbitrary numbers?
  - (1..9)(0..9)* | 0
- Java block comments?
  - /* (^* | (*+ ^/))* *+ /
- Here: characters and auxiliary symbols distinguished by colour
  - E.g., * above occurs both as character * and as Kleene star *


Kleene
1904 - 1994

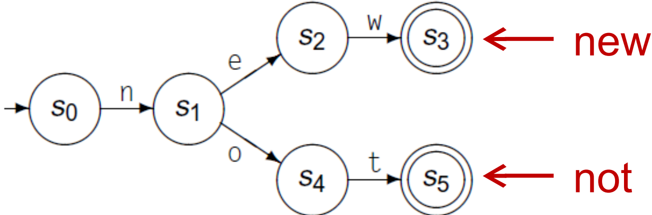
FROM RE'S TO DFA'S (SEE MODULE 2.3!)

- Automatic construction
  1. Construct Nondeterministic Finite Automaton (NFA) from the RE
  2. Determinize the NFA, yielding DFA (subset construction)
  3. Minimize the DFA
  4. Use DFA to accept/recognize tokens
- Step 2 may "blow up" the automaton
  - Every DFA state corresponds to a *set* of NFA states
  - If NFA has states, DFA may have how many?
    - states (= number of subsets of a set of elements)
- In the setting of scanners, this is never a problem in practice
  - Blow-up would occur if language has many similar tokens
  - This is avoided by design: also inconvenient for humans

> there are languages *designed* to be inconvenient: look up "esoteric languages" on Wikipedia

COMBINING MULTIPLE TOKEN TYPES

- A realistic language has many different token types
  - Java: 100 token types (e.g., every keyword is a different token type)
  - Every token type is given by an RE (and hence a DFA)
- Different DFA's can be combined into single "big" DFA for all tokens
  - How to distinguish between token types in big DFA?
    - Label the accepting states with corresponding token type



  - What if token types (i.e., DFA's) overlap?
    - E.g., token type for identifier overlaps with *every* keyword
    - Ambiguity: resolved by ordering of token types in spec

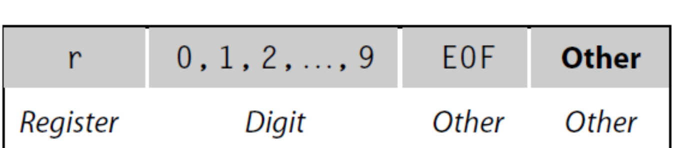
RECOGNIZING MULTIPLE TOKENS IN A ROW

- Input: character string *containing multiple tokens*
- Problem: we don't know where tokens are supposed to end
  - Not every token has a clear end marker
  - For instance: how should <== be scanned in Java?
    - < followed by == ?
    - <= followed by = ?
- Choice: *greedy* scanning algorithm
  - Even after recognizing a token, try to extend it to a longer one
  - If extending fails, backtrack to last token found
  - Never backtrack over an already accepted token!

GENERAL SCANNING ALGORITHM (MULTIPLE TOKENS)

```
NextWord()
  state ← s0;
  lexeme ← “ ”;
  clear stack;
  push(bad);

  while (state≠se) do              -- first proceed to error state
    NextChar(char);
    lexeme ← lexeme + char;
    if state ∈ SA
        then clear stack;
    push(state);                   -- remember states
    cat ← CharCat[char];
    state ← δ[state,cat];
  end;

  while(state ∉ SA and
        state≠bad) do              -- then roll back to last accepting state
    state ← pop();
    truncate lexeme;
    RollBack();
  end;
```

Example: r(0..9)+
(register names)

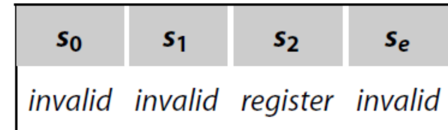


r	0, 1, 2, ..., 9	EOF	**Other**
Register	*Digit*	*Other*	*Other*

The Classifier Table, CharCat

	Register	*Digit*	*Other*
s_0	s_1	s_e	s_e
s_1	s_e	s_2	s_e
s_2	s_e	s_2	s_e
s_e	s_e	s_e	s_e

The Transition Table, δ

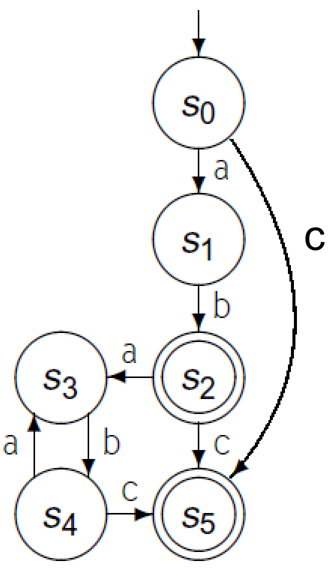


s_0	s_1	s_2	s_e
invalid	*invalid*	*register*	*invalid*

The Token Type Table, Type

ROLLING BACK

- Example: tokens ab and (ab)*c



- How is “abababab” scanned?
  - First try abababab as token (maybe a c will follow)
  - Discover failure
  - Roll back to ab
  - Then try ababab, again roll back to ab
  - Result: ab ab ab ab (as expected), but takes *quadratic* time
- Example: tokens (ab)+ and abc
  - How is “ababababc” scanned?
  - Scanner error (due to greedy scanning); *not* ababab abc

BOOK: MICRO-EFFICIENCY CONCERNS

- Considerations
  - Avoiding excess rollback
  - Optimizing lookup table access (not entire lookup table in memory)
  - Optimizing input buffers access (not entire program in one string)
  - Storing and comparing actual lexemes
- Variations on general scheduling algorithm
  - Direct-coded scanners
  - Hand-coded scanners
- Only for those interested

BUT:

- Realise that if `s` is a string, then
  ```
  s = s + "a";
  ```
  takes time *linear in the length of* `s` (due to copying)

SEEN TODAY

- Deterministic Finite Automata
  - General acceptance algorithm (single token)
- Regular expressions
  - Conversion to DFA's
- Table-driven scanner
  - General scanning algorithm (multiple tokens)

- Lab exercises:
  - Play around with RE's and DFA's
  - Implement and test the acceptance and scanning algorithms