# Compiler Construction

## Scanning
## Lexical Analysis

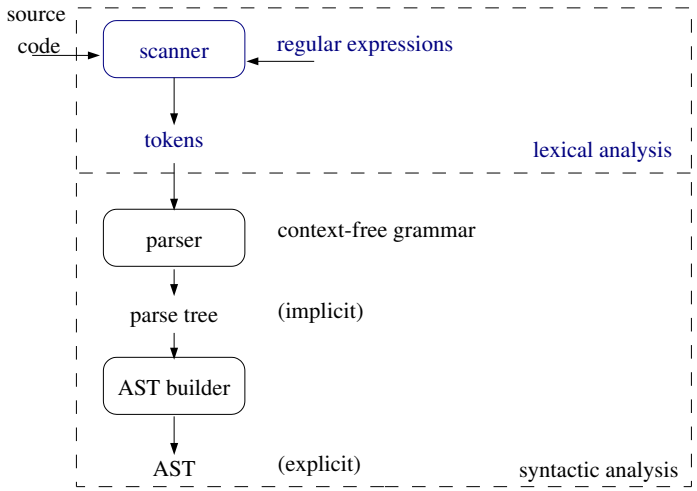
Lennart Andersson

Revision 2012–01–18

2012



## Compiler phases






## Typical tokens

| | lexemes |
|---|---|
| Reserved words (keywords) | if then else begin |
| Identifiers | i alpha k10 |
| Literals | 123 3.1416 'A' "text" |
| Operators | + ++ != |
| Separators | ; , ( |



## Non-tokens

| | lexemes |
|---|---|
| whitespace | ' ' tab newline formfeed |
| comments | /* comment */ //eol comment |

## Language concepts

- An *alphabet* is a nonempty finite set of *symbols*.
- A *string* is a finite sequence of symbols.
- A *language* is a set of strings.

$L_1 L_2$ contains all strings $s_1 s_2$ where $s_1 \in L_1$ and $s_2 \in L_2$



## Regular expression usage

- grep, egrep, fgrep: unix search utilities
- emacs: re-search-forward, replace-regexp
- sed: stream editor with re capabilities
- java.util.regex pattern matching classes



## Utility scanners

- java.io.StreamTokenizer
- java.util.Scanner
- java.util.regex



## Regular expressions, canonical

$\mathcal{L}[\![M]\!]$ is the language described by the regular expression $M$.

| expression $M$ | language $\mathcal{L}[\![M]\!]$ |
|---|---|
| $\emptyset$ | $\{\}$ |
| a | $\{$ "a" $\}$ |
| $\alpha \mid \beta$ | $\mathcal{L}[\![\alpha]\!] \cup \mathcal{L}[\![\beta]\!]$ |
| $\alpha \cdot \beta$ | $\mathcal{L}[\![\alpha]\!]\mathcal{L}[\![\beta]\!]$ |
| $\alpha^*$ | $\{$ "" $\} \cup \mathcal{L}[\![\alpha]\!] \cup \mathcal{L}[\![\alpha \cdot \alpha]\!] \cup \mathcal{L}[\![\alpha \cdot \alpha \cdot \alpha]\!] \cup \ldots$ |
| $(\alpha)$ | $\mathcal{L}[\![\alpha]\!]$ |

$\alpha$ and $\beta$ denote regular expressions.

We will use $\epsilon$ as an abbreviation of $\emptyset^*$. $\mathcal{L}[\![\epsilon]\!] = \{""\}$.

## Operator precedence (binding power)

| operator | precedence |
|---|---|
| $^*$ | highest |
| $\cdot$ | |
| $\mid$ | lowest |

| expression | interpretation | language |
|---|---|---|
| $a \cdot b^*$ | $a \cdot (b^*)$ | |
| $a \cdot b \mid c$ | $(a \cdot b) \mid c$ | |
| $a \cdot b^* \mid c$ | $(a \cdot (b^*)) \mid c$ | |

$\cdot$ and $\mid$ are associative.
$a \cdot (b \cdot c)$ is equivalent to $(a \cdot b) \cdot c$;
they denote the same language.



## Activity

Write a regular expression describing the language of all binary strings

1. that ends with 11.
2. that does not end with 11.
3. with an even number of 1's

Write a regular expression describing the language of all arithmetic expressions with binary numbers and the operators + and *, but without parentheses.



## Extended regular expressions

| Appel | JavaCC | Canonical RE |
|---|---|---|
| $\epsilon$ | | $\emptyset^*$ |
| $MN$ | M N | M·N |
| $M^+$ | M+ | M·M$^*$ |
| $M?$ | M? | $\emptyset^*$ \| M |
| $[a-zA-Z]$ | [ "a" - "z" , "A" - "Z" ] | a\|b\| etc. |
| | ~[ "0" – "9" ] | (*too long to fit*) |
| | "axb" | a·x·b |
| \n | "\n" | |



## JavaCC example . . .

```
/* Skip whitespace */
SKIP : { " " | "\t" | "\n" | "\r" }

/* Reserved words */
TOKEN [IGNORE_CASE]: {
< IF: "if">
| < THEN: "then">
}

/* Identifiers */
TOKEN: {
< ID: (["A"-"Z", "a"-"z"])+ >
}
```

## . . . JavaCC example

```
/* Literals */
TOKEN: {
< INT: (["0"-"9"])+ >
}

/* Operators and separators */
TOKEN: {
< ASSIGN: "=" >
| < MULT: "*" >
| < LT: "<" >
| < LE: "<=" >
...
}
```



## Ambiguous lexical definitions

The scanned text can match tokens in more than one way

- Which tokens match `"<="` ?
- Which tokens match `"if"` ?



## Extra rules for resolving the ambiguities

Longest match

- If one rule can be used to match a token, but there is another rule that will match a longer token, the latter rule will be chosen. This way, the scanner will match the longest token possible.

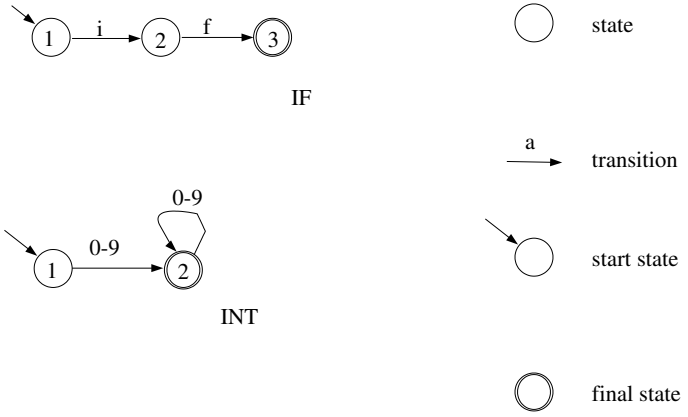
Rule priority

- If two rules can be used to match the same sequence of characters, the first one takes priority.



## Implementation of scanners

Finite automata



## More automata

WHITESPACE

ID



## Automaton for the whole language

Combine the automata for individual tokens

- merge the start states
- re-enumerate the states so they get unique numbers
- mark each final state with the token that is matched



## Example 1

Combine IF and INT



## Example 2

Combine IF and ID

## Translating a RE to an NFA

- $a$
- $MN$
- $M \mid N$
- $M^*$
- $\epsilon$



## DFA vs. NFA

Deterministic Finite Automaton (DFA)
A finite automaton is deterministic if

- all outgoing edges from any given node have disjoint character sets
- there are no $\epsilon$ transitions (the empty string)

Can be implemented efficiently

Non-deterministic Finite Automaton (NFA)

An NFA may have

- two outgoing edges with overlapping character sets
- $\epsilon$-edges

DFA $\subset$ NFA



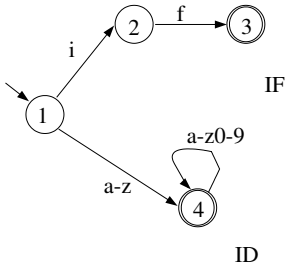
## Each NFA can be translated to a DFA

- Each state in the DFA will correspond to a set of states in the NFA
- There is a path in the NFA from state $s$ to state $t$ consuming the symbol $a$ and any number $\epsilon$ transitions there will be an $a$ transition from any state containing $s$ to any state containing $t$ in the DFA.
- The start state of the DFA will contain the start state of the NFA and all states in the NFA reachable from the start state via $\epsilon$ transitions ($\epsilon$ closure).
- Every state in the DFA containing a final state in the NFA will be a final state labeled with the same token.



## Combine IF and ID

NFA DFA



## Construction of a Scanner

- Define all tokens as regular expressions
- Construct a finite automaton for each token
- Combine all these automata to a new automaton (an NFA in general)
- Translate to a corresponding DFA
- Minimize the DFA
- Implement the DFA



## Implementation alternatives for DFAs

Table-driven

- Represent the automaton by a table
- Additional table to keep track of final states and token kinds
- A global variable keeps track of the current state

Switch statements

- Each state is implemented as a switch statement
- Each case implements a state transition as a jump (to another switch statement).
- The current state is represented by the program counter



## Table-driven implementation

DFA for IF and ID (F2-23)

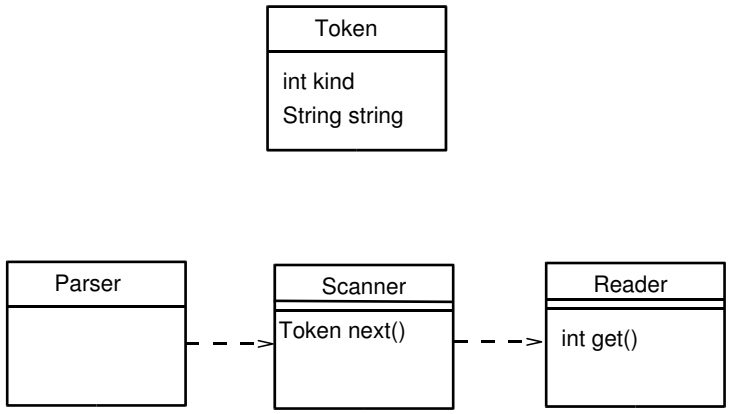
| | .. | a-e | f | g-h | i | j-z | .. | final | kind |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | true | ERROR |
| 1 | | | | | | | | | |
| 2,4 | | | | | | | | | |
| 3,4 | | | | | | | | | |
| 4 | | | | | | | | | |



## Error handling

- Add transitions to the 'dead state' (state 0) for all erroneous input
- Generate an 'Error token' when the dead state is reached

## Design






## Scanner (sketch)

```
class Scanner {
  PushbackReader reader;
  boolean isFinal [ ];
  int edges [ ] [ ];
  int kind [ ];
  Token next() {









  }
}
```



## Handling End-Of-File (EOF) and non-tokens

EOF

- construct an explicit EOF token when the EOF character is read

Non-tokens (Whitespace & Comments)

- view as tokens of a special kind
- scan them as normal tokens, but don't create token objects for them
- loop in `next()` until a real token has been found

Errors

- construct an explicit ERROR token to be returned when no valid token can be found.



## Handling "longest match"

- When a token is matched (a final state reached), don't stop scanning.
- Keep track of the latest matched token in a variable.
- Continue scanning until we reach the "dead state"
- Restore the input stream. Use PushbackReader to be able to do this.
- Return the latest matched token
- (or return the ERROR token if there was no latest matched token)

## Scanner with longest match

```
class Scanner {
  ...
  Token next() {









  }
}
```

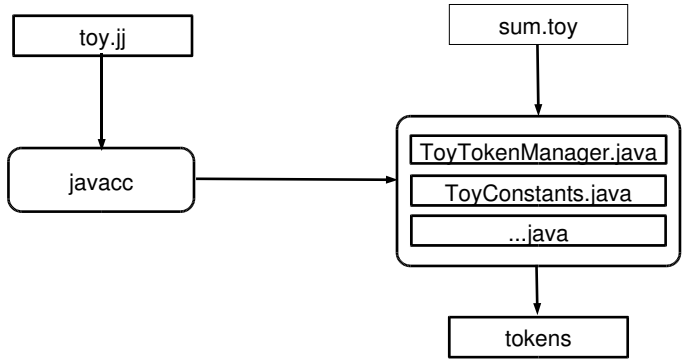


## Other scanner generators

| tool | author | implementation | generates |
|---|---|---|---|
| lex | Schmidt, Lesk, 1975 | table-driven | C code |
| flex | Paxon. 1987 | switch statements | C code |
| jlex | | table-driven | Java code |
| jflex | | switch statements | Java code |



## JavaCC as a scanner generator






## Additional functionality

Lexical actions

- for adapting the generated scanner
- e.g., create special token objects, count tokens, keep track of position in input text (line and column numbers), etc.

Lexical states ('start states')

- gives the possibility to switch automata during scanning

## Summary questions

- What is meant by an ambiguous lexical definition?
- Give some typical examples of this and how they may be resolved.
- Construct an NFA for a given lexical definition
- Construct a DFA for a given NFA
- What is the difference between a DFA and an NFA?
- Implement a DFA in Java.
- How is rule priority handled in the implementation? How is longest match handled? EOF? Whitespace?



## Readings

F2: Scanning. Regular expressions, deterministic and nondeterministic finite automata. Scanning with JavaCC.

- Appel, chapter 2-2.4
- Hedin, Quick guide to writing scanning definitions in JavaCC, `www.cs.lth.se/EDA180/2007/tools/scanningguide.html`

F3: Parsing. Context-free grammars. Predictive parsing, Parsing with JavaCC.

- Appel, chapter 3-3.2
- JavaCC documentation, `javacc.dev.java.net/doc/docindex.html`

S1: Seminar 1. Lexical Analysis.

- Study F1, F2, and associated readings.

L1: Programming assignment 1. Scanning

- Study the lab material