# Programming Languages and Compilers

## Symbol Specification and Lexical Analysis

Dr. Peter Pfahler
Based on the lecture by Prof. Dr. Uwe Kastens

Universität Paderborn
Fakultät EIM
Institut für Informatik

Winter 2016/2017



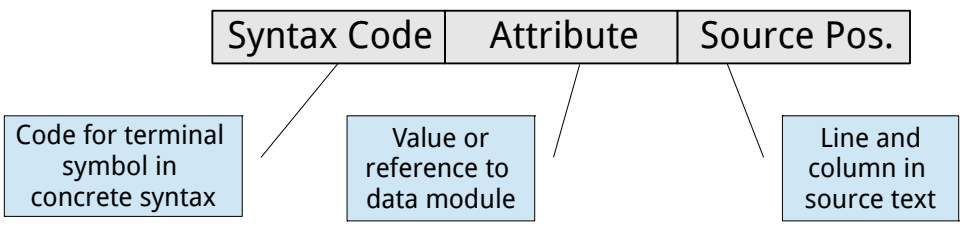


## Lexical Analysis

**Input:** Program represented by a sequence of characters

| **Tasks** | **Compiler Module** |
|---|---|
| Read character stream | Input reader |
| Recognize and classify tokens; skip irrelevant characters | Scanner (central phase, finite state machine) |
| Encode tokens; Conversion; Store token information | Identifier Module, Literal Module |

**Output:** Program represented by a sequence of *tokens* encoded by triples:



## Token Representation

### Example SetLan Program

```
{
  set one;
  set two;
  one = [100];
  two = one + [77];
}
```

| | | | |
|---|---|---|---|
| 5 | 0 | (1, 1) | { |
| 35 | 0 | (2, 3) | set |
| 36 | 2 | (2, 7) | one |
| 14 | 0 | (2, 10) | ; |
| 35 | 0 | (3, 3) | set |
| 36 | 3 | (3, 7) | two |
| 14 | 0 | (3, 10) | ; |
| 36 | 2 | (4, 3) | one |
| 32 | 0 | (4, 7) | = |
| 21 | 0 | (4, 9) | [ |
| 26 | 100 | (4, 10) | 100 |
| 22 | 0 | (4, 13) | ] |
| 14 | 0 | (4, 14) | ; |
| 36 | 3 | (5, 3) | two |
| 32 | 0 | (5, 7) | = |
| 36 | 2 | (5, 9) | one |
| 4 | 0 | (5, 13) | + |
| 21 | 0 | (5, 15) | [ |
| 26 | 77 | (5, 16) | 77 |
| 22 | 0 | (5, 18) | ] |
| 14 | 0 | (5, 19) | ; |
| 6 | 0 | (6, 1) | } |
| 1 | 0 | (7, 1) | EOF |

Typical token classes are

- Identifiers
- Keywords
- Literals
- Special Symbols

Attribute values are relevant for identifiers and literals.

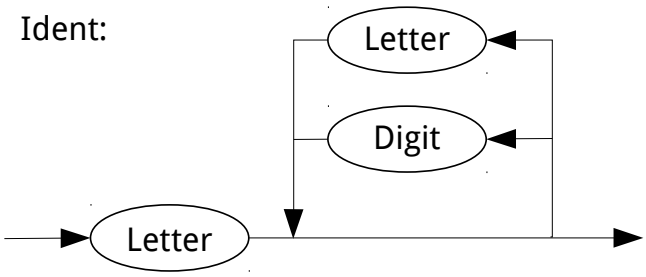




## Token Specification and Recognition

```
Ident :
   Letter (Letter | Digit)*
```

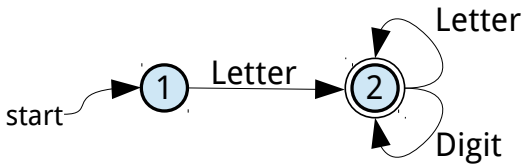
**Typical Notation:**
**Regular Expression**




**Graphical Notation:**
**Syntax Diagram**

```
Ident : Letter X
X     : Letter X
X     : Digit X
X     :
```

**Formal Notation:**
**Regular Grammar**

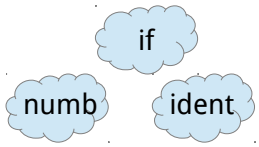



**Implementation:**
**Finite Automaton**

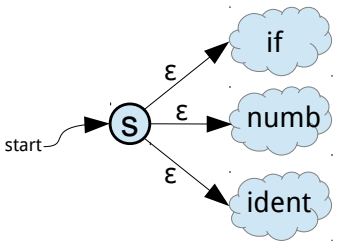
## Constructing Scanners from Token Specifications

The scanner is implemented as a finite automaton constructed as follows:

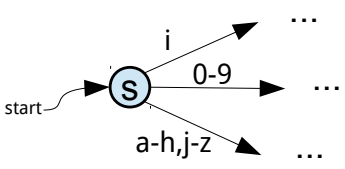
- For each token specification map the regular expression to a nondeterministic finite automaton (NFA)




- Unite the starting states of all automata



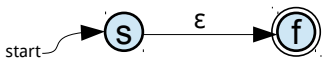
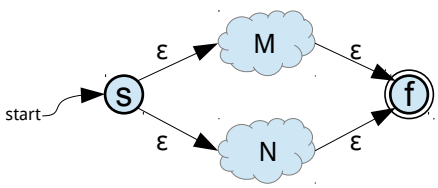
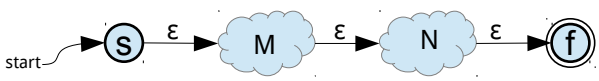
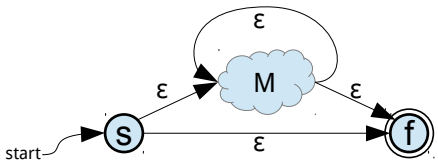

- Transform the NFA into a deterministic finite automaton (DFA)








## Regular Expressions mapped to Nondeterministic Finite Automata

| **Regular Expression** | **NFA** |
|---|---|
| ε |  |
| a |  |
| M \| N |  |
| M N |  |
| M* |  |



**Shorthands:**

M+ is constructed as: `M M*`

M? is constructed as: `M | ε`

`[abc]` is constructed as: `a | b | c`

## Transform the NFA into a DFA (1)

The "subset construction" uses two functions:

- The $\varepsilon$-*closure* function takes a state and returns the set of states reachable from it based on (one or more) $\varepsilon$-transitions.
- The function *move* takes a state and a character, and returns the set of states reachable by one transition on this character.

These functions are generalized to apply to sets of states by taking the union of the application to individual states.

**Example:**

If A, B and C are states,
then $move(\{A,B,C\},\text{'a'}) = move(A,\text{'a'}) \cup move(B,\text{'a'}) \cup move(C,\text{'a'})$





## Transform the NFA into a DFA (2)

**The Subset Construction Algorithm**

1. Create the start state of the DFA by taking the NFA start state's $\varepsilon$-*closure* .
2. Perform the following for the new DFA state:
   For each possible input symbol:
   - Apply *move* to the newly-created state and the input symbol; this will return a set of states.
   - Apply the $\varepsilon$-*closure* to this set of states, possibly resulting in a new set.

   This set of NFA states will be a single state in the DFA.
3. Each time we generate a new DFA state, we must apply step 2 to it.
   The process is complete when applying step 2 does not yield any new states.
4. The final states of the DFA are those which contain any of the NFA's final states.

Example: Construct the DFA for the following token set:

```
IFSYMB : if
IDENT  : [a-z] ([a-z] | [0-9])*
NUMB   : [0-9]+
```

## Recognize the Longest Match

An automaton may contain transitions from final states: When does the automaton stop?

### Rule of the longest match

- The automaton continues as long as there is a transition with the next character.
- After having stopped it sets back to the most recently passed final state.
  The input position is adjusted.
- If no final state has been passed an error message is issued.

Consequence: Some kinds of tokens have to be separated explicitly.





## Aspects of Scanner Implementation

Scanner runtime is proportional to the number of characters in the program
$\Rightarrow$ Operations per character must be fast, otherwise the scanner dominates compilation time.

Two implementation techniques:

- Table driven scanning:
  Scanner control loop interprets state transition table.
- Directly programmed scanning:
  State transitions are coded by control flow (switches, branches, loops).

Directly programmed scanner implementations are usually faster than table-driven scanners.

## Aspects of Lexical Level Language Design

Tokens should be **recognized in isolation**:

e. G. all occurrences of the identifier `a` get the same encoding:

```
{int a; ... a = 5; ... {float a; ... a = 3.1; ...}}
```

distinction of the two different variables would require information from semantic analysis

**typedef problem in C**:

The C syntax requires **lexical distinction** of type-names and other names:

```
typedef int *T; T (*B); X (*Y);
```

cause syntactically different structures: declaration of variable `B` and call of function `X`.
Requires feedback from semantic analysis to lexical analysis.

Identifiers in PL/1 may **coincide with keywords**:

```
if if = then then := else else := then
```

Lexical analysis needs feedback from syntactic analysis to distinguish them.

**Token separation** in FORTRAN:

„Deletion or insertion of blanks does not change the meaning.“

```
DO 24 K = 1,5      begin of a loop, 7 tokens
DO 24 K = 1.5      assignment to the variable DO24K, 3 tokens
```

Token separation is determined late.





## Identifier and Literal Modules

- Uniform interface for all scanner support modules:
  - Input parameters: pointer to token text and its length;
  - Output parameters: syntax code, attribute
- Identifier module (aka “Symbol Table”) uniquely encodes identifier occurrences
  Implementation: hash table
  Possibly also encodes keywords
- Literal modules for floating point numbers, integral numbers, strings
  Variants for representation in memory:
  - token text
  - value converted into compile site representation
  - value converted into target site representation

Caution: Avoid overflow on conversion!
Cross compiler: compiler representation may differ from target representation

## Scanner Generators

Scanner generators generate the central function of lexical analysis from token specifications given as regular expressions.

- Lex / Flex
  - standard Unix tool and its successor
  - scanner implemented in C/C++
  - table driven
- GLA (Generator for Lexical Analysis)
  - University of Boulder, Colorado
  - Part of the Eli system, interfaces with other components
  - library of RE for typical tokens ("canned descriptions")
  - scanner implemented in C/C++
  - directly programmed implementation
- JLex / JFlex
  - Java Scanner Generator and its successor, GNU license
  - scanner implemented in Java
  - table driven





## Example: SetLan Token Specification for Eli

Written in GLA notation using canned descriptions.
Literal tokens (keywords and special symbols) are automatically taken from concrete syntax and do not have to be specified in Eli.

**SetLan Token Specification**

```
Identifier: C_IDENTIFIER
Number:     C_INTEGER
            C_COMMENT
```

Equivalent specification after expanding canned descriptions:

**SetLan Token Specification without Canned Descriptions**

```
Identifier: $[a-zA-Z_][a-zA-Z_0-9]* [mkidn]
Number:     $([0-9]+|0[xX][0-9a-fA-F]*) [c_mkint]
            $"/*" (auxCComment)
```

The specification uses *token processors* like `mkidn` to access identifier and literal modules and so-called *auxilliary scanners* like `auxCComment` for special scanning purposes, like e.g. skipping (possibly nested) comments.