Compilerconstructie

najaar 2018

`http://www.liacs.leidenuniv.nl/~vlietrvan1/coco/`

Rudy van Vliet
kamer 140 Snellius, tel. 071-527 2876
rvvliet(at)liacs(dot)nl

college 2, vrijdag 14 september 2018

Symbol Table / Lexical Analysis

2.7 Symbol Table

- Symbol table holds information about *source-program constructs* (e.g., identifiers)
  - string
  - additional information (type, position in storage, . . .)

- Symbol table is globally accessible (to all phases of compiler)

- Information is collected incrementally by analysis phases, and used by synthesis phases

- (Possible) implementation by Hashtable, with methods
  - *put (String, Symbol)*
  - *get (String)*

Symbol Table Per Scope

The same identifier may be declared more than once

```
{ int x; int y;
  { int w; bool y; int z;
    ... w ...; ... x ...; ... y ...; ... z ...;
  }
  ... w ...; ... x ...; ... y ...;
}
```

Symbol Table Per Scope

The same identifier may be declared more than once

```
for (int i=1;i<=3;i++)
{ for (int i=1;i<=5;i++)
    cout << "Hello inner world" << endl;
  cout << "  Hello outer world" << endl;
}
```

Symbol Table Per Scope

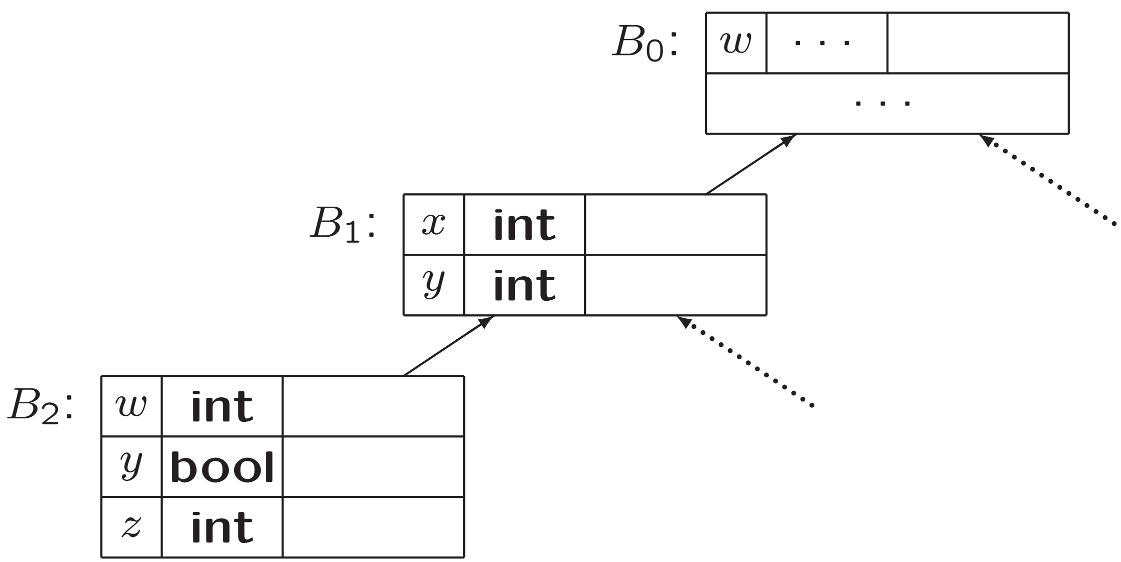
The same identifier may be declared more than once

```
Hello inner world
Hello inner world
Hello inner world
Hello inner world
Hello inner world
  Hello outer world
Hello inner world
Hello inner world
Hello inner world
Hello inner world
Hello inner world
  Hello outer world
Hello inner world
Hello inner world
Hello inner world
Hello inner world
Hello inner world
  Hello outer world
```

Symbol Table Per Scope

The same identifier may be declared more than once

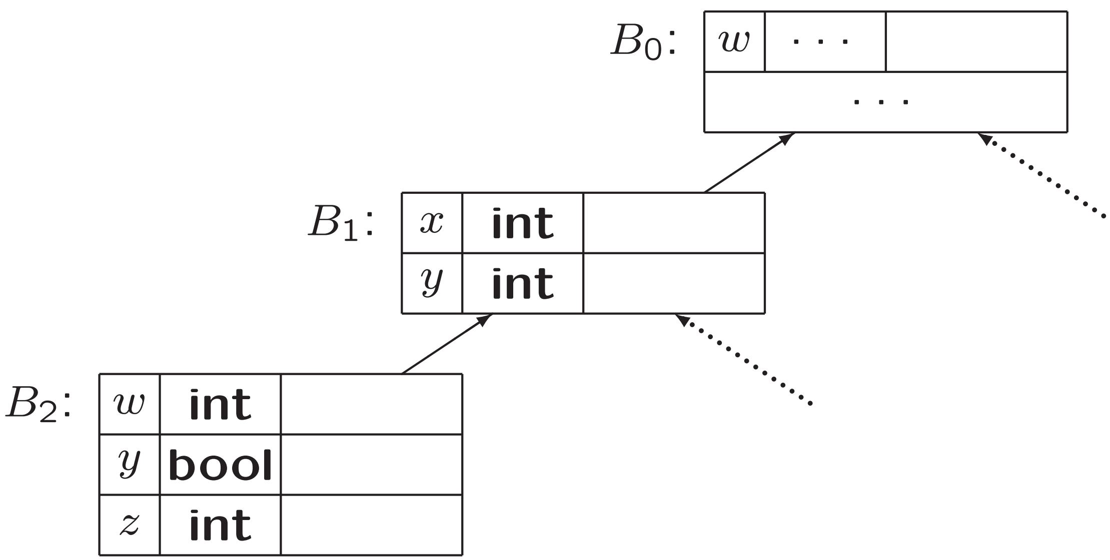
```
1)  { int x1; int y1;
2)    { int w2; bool y2; int z2;
3)      ... w2 ...; ... x1 ...; ... y2 ...; ... z2 ...;
4)    }
5)    ... w0 ...; ... x1 ...; ... y1 ...;
6)  }
```



Symbol Table Per Scope

The same identifier may be declared more than once

```
{ int x1; int y1;
  { int w2; bool y2; int z2;
    ... w2 ...; ... x1 ...; ... y2 ...; ... z2 ...;
  }
  ... w0 ...; ... x1 ...; ... y1 ...;
}
```



Symbol tables per block can be allocated and deallocated in stack-like fashion, but. . .

Implementation Symbol Table

(in Java)

```
public class Env
{ private Hashtable table;
  protected Env prev;

  public Env (Env p)
  { table = new Hashtable();
    prev = p;
  }

  public void put (String s, Symbol sym)
  { table.put (s, sym);
  }

  public Symbol get (String s);
  { for (Env e=this; e!=null; e=e.prev)
    { Symbol found = (Symbol)(e.table.get(s));
      if (found != null)
        return found;
    }
    return null;
  }
}
```

Translation Scheme (Example)

(from lecture 1)

$$
\begin{array}{rcl}
\textit{expr} & \rightarrow & \textit{expr}_1 + \textit{term}\ \{\text{print('+')}\} \\
\textit{expr} & \rightarrow & \textit{expr}_1 - \textit{term}\ \{\text{print('}-\text{')}\} \\
\textit{expr} & \rightarrow & \textit{term} \\
\textit{term} & \rightarrow & 0\ \{\text{print('0')}\} \\
\textit{term} & \rightarrow & 1\ \{\text{print('1')}\} \\
\ldots & & \ldots \\
\textit{term} & \rightarrow & 9\ \{\text{print('9')}\}
\end{array}
$$

Example: parse tree for $9 - 5 + 2 \ldots$

Implementation requires postorder traversal (LRW)

CFG for Program with Blocks

$$
\begin{array}{lcl}
\textit{program} & \rightarrow & \textit{block} \\
\textit{block} & \rightarrow & '\{' \textit{decls stmts} '\}' \\
\textit{decls} & \rightarrow & \textit{decls decl} \\
 & | & \epsilon \\
\textit{decl} & \rightarrow & \textbf{type id}; \\
\textit{stmts} & \rightarrow & \textit{stmts stmt} \\
 & | & \epsilon \\
\textit{stmt} & \rightarrow & \textit{block} \\
 & | & \textit{factor};
\end{array}
$$

The Use of Symbol Tables

$$
\begin{array}{lcll}
\textit{program} & \rightarrow & & \{\ \textit{top} = \textbf{null};\ \} \\
 & & \textit{block} & \\
 & & & \\
\textit{block} & \rightarrow & '\{' & \{\ \textit{saved} = \textit{top}; \\
 & & & \quad \textit{top} = \textbf{new}\ \textit{Env}(\textit{top}); \\
 & & & \} \\
 & & \textit{decls}\ \textit{stmts}\ '\}' & \{\ \textit{top} = \textit{saved}; \\
 & & & \} \\
 & & & \\
\textit{decls} & \rightarrow & \textit{decls}\ \textit{decl} & \\
 & | & \epsilon & \\
 & & & \\
\textit{decl} & \rightarrow & \textbf{type}\ \textbf{id}; & \{\ s = \textbf{new}\ \textit{Symbol}; \\
 & & & \quad s.\textit{type} = \textbf{type}.\textit{lexeme}; \\
 & & & \quad \textit{top.put}(\textbf{id}.\textit{lexeme}, s); \\
 & & & \}
\end{array}
$$

In book (edition 2) extended for real translation

2.6 Lexical Analyser

Reads and converts the input into a stream of tokens to be analysed by the parser

Lexeme: Sequence of input characters comprising single token

Typical tasks of the lexical analyser

- Remove white space and comments
- Encode constants as tokens:

  $$31 + 28 + 59 \rightarrow \langle \textbf{num}, 31 \rangle \ \langle + \rangle \ \langle \textbf{num}, 28 \rangle \ \langle + \rangle \ \langle \textbf{num}, 59 \rangle$$

- Recognize keywords
- Recognize identifiers:

  $$\texttt{count} = \texttt{count} + \texttt{increment;} \quad \rightarrow$$
  $$\langle \textbf{id}, ''\texttt{count}'' \rangle \ \langle = \rangle \ \langle \textbf{id}, ''\texttt{count}'' \rangle \ \langle + \rangle \ \langle \textbf{id}, ''\texttt{increment}'' \rangle \ \langle ; \rangle$$

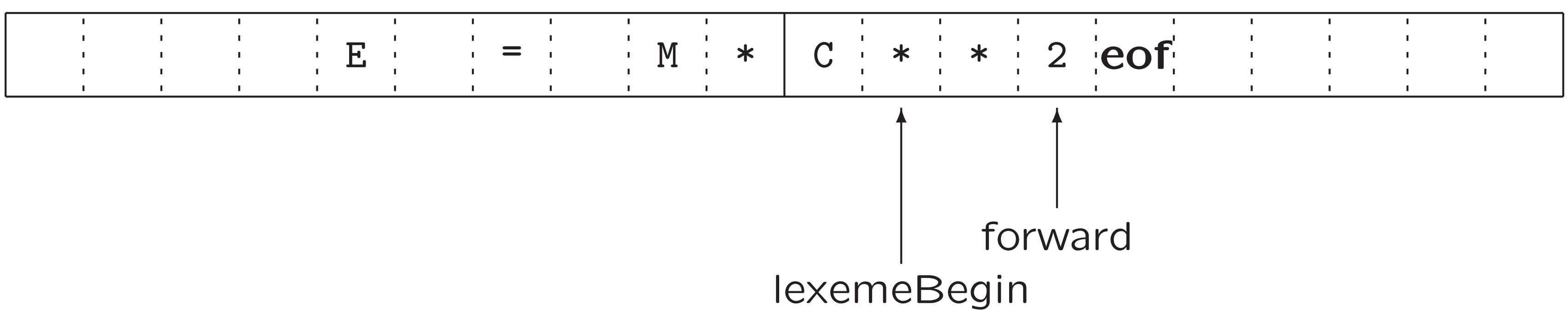
Lexical analyser may need to read ahead (with input buffer)

3.2 Input Buffering

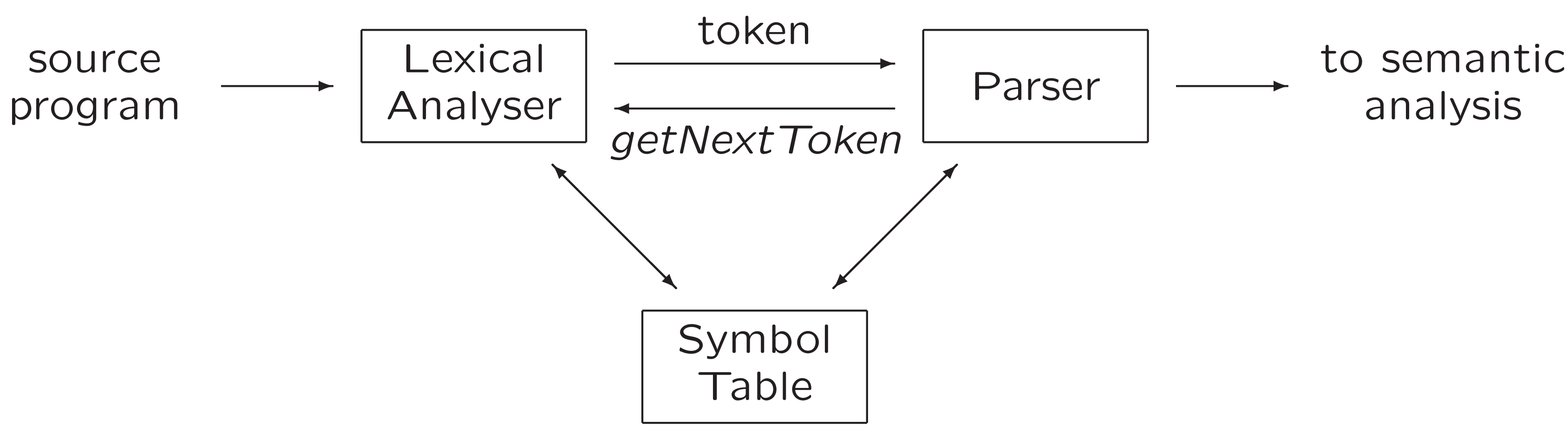
Use two buffers of size N for input

- Saves time
- Allows for looking ahead one or more characters, e.g., for
  - identifiers: `ifoundit`
  - relational operators: `<=`

Take longest prefix of input that matches any pattern



3.1 Lexical Analyser - Parser Interaction



3.1.2 Tokens, Patterns and Lexemes

- Token: pair of token name and optional attribute value, e.g., $\langle \textbf{id}, 1 \rangle, \langle \textbf{num}, 31 \rangle, \langle \textbf{assign\_op} \rangle$

- Lexeme: specific sequence of characters that makes up token, e.g., `count`, `31`, `=`

- Pattern: description of form that lexemes of a token may take

Examples of Tokens

Token	Informal Description	Sample Lexemes
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
literal	anything but `"`, surrounded by `"`'s	`"core dumped"`

3.1.3 Attributes for Tokens

`E = M * C ** 2`

↓

Lexical Analyser

↓

⟨**id**, pointer to symbol-table entry for `E`⟩ ⟨**assign_op**⟩
⟨**id**, pointer to symbol-table entry for `M`⟩ ⟨**mult_op**⟩
⟨**id**, pointer to symbol-table entry for `C`⟩ ⟨**exp_op**⟩
⟨**number**, integer value 2⟩

↓

Parser

3.1.4 Lexical Errors

- Hard to detect by lexical analyser alone, e.g.,

```
fi ( a == f(x) ) ...
```

- What if none of the patterns matches?
  - 'Panic mode' recovery: delete characters until you find well-formed token
  - ∗ Delete one character from remaining input
    ∗ Insert missing character into remaining input
    ∗ Replace character by another character
    ∗ Transpose two adjacent characters

Implementing a Lexical Analyser

- By hand,
  using transition diagram to specify lexemes

- With a lexical-analyser generator (`Lex`),
  using regular expressions to specify lexemes:

  Regular expressions $\rightarrow$
      (non-deterministic) finite automaton $\rightarrow$
      determininistic finite automaton
  Input to 'driver'

4.3.1 Why Regular Expressions For Lexical Syntax?

Instead of adding rules to grammar

- Convenient way to modularize front end
  $\approx$ simplifies design

- Regular expressions powerful enough for lexical syntax

- Regular expressions easier to understand than grammars

- More efficient lexical analysers can be constructed automatically from regular expressions than from arbitrary grammars

3.1.1 Lexical Analyser

Reasons why it is a separate phase of a compiler

- Simplifies the design of the compiler

- Provides efficient implementation
  - Systematic techniques to implement lexical analysers (by hand or automatically)

- Improves portability
  - Non-standard symbols and alternate character encodings can be more easily translated (only relevant for lexical analyser)

3.3 Specification of Tokens

Regular expressions to specify patterns for tokens

Terminology (from FI1)

- An alphabet Σ is a finite set of symbols (characters), e.g., $\{0, 1\}$, ASCII, Unicode

- A string s is a finite sequence of symbols from Σ
  - $|s|$ denotes the length of string s, e.g., $|\text{banana}| = 6$
  - ϵ denotes an empty string: $|\epsilon| = 0$

- A language is a set of strings over some fixed alphabet Σ

String operations

- Concatenation of strings x and y is denoted as xy
  e.g., if $x = \texttt{dog}$ and $y = \texttt{house}$ then $xy = \texttt{doghouse}$
  $s\epsilon = \epsilon s = s$

- Exponentiation
  - Define

$$\begin{aligned} s^0 &= \epsilon \\ s^i &= s^{i-1}s \quad \text{if } i > 0 \end{aligned}$$

  - Then

$$\begin{aligned} s^1 &= s \\ s^2 &= ss \\ s^3 &= sss \end{aligned}$$

Language Operations

- Union $\quad L \cup D = \{s \mid s \in L \text{ or } s \in D\}$

- Concatenation $\quad LD = \{xy \mid x \in L \text{ and } y \in D\}$

- Exponentiation $\quad L^0 = \{\epsilon\}; \qquad L^i = L^{i-1}L \quad \text{if } i > 0$

- Kleene closure $\quad L^* = \cup_{i=0,\ldots,\infty} L^i$
  (zero or more concatenation)

- Positive closure $\quad L^+ = \cup_{i=1,\ldots,\infty} L^i$
  (one or more concatenation)

Language Operations (Example)

Let alphabets $L = \{\mathtt{A}, \mathtt{B}, \ldots, \mathtt{Z}, \mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$ and $D = \{0, 1, \ldots, 9\}$

- $L \cup D$ is set of letters and digits
- LD is set of strings consisting of a letter followed by a digit
- L^4 is set of all four-letter strings
- L^* is set of all finite strings of letters, including ϵ
- $L(L \cup D)^*$ is set of all strings of letters and digits beginning with a letter ('identifiers')
- D^+ is set of all strings of one or more digits ('nonnegative integers')

Regular Expressions (Example)

In C, an identifier is a letter followed by zero or more letters or digits (underscore is considered letter):

$$\mathit{letter\_}\ (\mathit{letter\_}\ \mid \mathit{digit}\ )^*$$

Regular Expressions (Definition)

- Each regular expression r denotes a language $L(r)$

- Defining rules:
  - ϵ is regular expression, and $L(\epsilon) = \{\epsilon\}$
  - if $a \in \Sigma$, then **a** is regular expression, and $L(\mathbf{a}) = \{a\}$.
  - if r and s are regular expressions, then
    - $(r) \mid (s)$ is regular expression denoting $L(r) \cup L(s)$
    - $(r)(s)$ is regular expression denoting $L(r)L(s)$
    - $(r)^*$ is regular expression denoting $(L(r))^*$
    - (r) is regular expression denoting $L(r)$

Regular Expressions (Example)

- Remove unnecessary parentheses by assuming precedence relation between $*$, concatenation, and $|$, e.g.,

  $(\mathbf{a}) \mid ((\mathbf{b})^*(\mathbf{c}))$ is equivalent to $\mathbf{a} \mid \mathbf{b}^*\mathbf{c}$

- Let $\Sigma = \{a, b\}$. Then the regular expression:
  - $\mathbf{a} \mid \mathbf{b}$ denotes the set $\{a, b\}$
  - $(\mathbf{a} \mid \mathbf{b})(\mathbf{a} \mid \mathbf{b})$ denotes the set $\{aa, ab, ba, bb\}$
  - $\mathbf{a}^*$ denotes the set $\{\epsilon, a, aa, aaa, \ldots\}$
  - $(\mathbf{a} \mid \mathbf{b})^*$ denotes the sets of all strings over $\{a, b\}$
  - $\mathbf{a} \mid \mathbf{a}^*\mathbf{b}$ denotes the string a and all strings consisting of zero or more a's followed by one b

- If r and s denote the same language L, then $r = s$, e.g., $(\mathbf{a} \mid \mathbf{b}) = (\mathbf{b} \mid \mathbf{a})$

Regular Expressions (Example)

In C, an identifier is a letter followed by zero or more letters or digits (underscore is considered letter):

$$\mathit{letter}_{\_}\ (\mathit{letter}_{\_}\ \mid \mathit{digit}\ )^{*}$$

Regular Definitions (Example)

- Identifiers in C

$$\begin{aligned} \mathit{letter\_} &\rightarrow \mathtt{A} \mid \mathtt{B} \mid \ldots \mid \mathtt{Z} \mid \mathtt{a} \mid \mathtt{b} \mid \ldots \mid \mathtt{z} \mid \_ \\ \mathit{digit} &\rightarrow 0 \mid 1 \mid \ldots \mid 9 \\ \mathit{id} &\rightarrow \mathit{letter\_}(\mathit{letter\_} \mid \mathit{digit})^{*} \end{aligned}$$

Regular Definitions

- A regular definition is a sequence of definitions of the form:

$$\begin{aligned} d_1 &\rightarrow r_1 \\ d_2 &\rightarrow r_2 \\ &\dots \\ d_n &\rightarrow r_n \end{aligned}$$

  where r_i is a regular expression over $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$

- Obtain regular expression over Σ by ...

Regular Definitions

- A regular definition is a sequence of definitions of the form:

$$
\begin{aligned}
d_1 &\rightarrow r_1 \\
d_2 &\rightarrow r_2 \\
&\cdots \\
d_n &\rightarrow r_n
\end{aligned}
$$

  where r_i is a regular expression over $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$

- Obtain regular expression over Σ by substituting $d_1, \ldots, d_{i-1}$ in r_i by $r_1, \ldots, r_{i-1}$ $(i = 2, \ldots, n)$

Regular Definitions (Example)

- Recursion is not allowed

$$
\begin{array}{rcll}
\mathit{digit} & \rightarrow & \mathit{digit}(\mathit{digit})^* & \text{not OK} \\
\mathit{digits} & \rightarrow & \mathit{digit}(\mathit{digit})^* & \text{OK}
\end{array}
$$

Notational Shorthands

- We often use the following shorthands:
  - one-or-more instance of: $r^{+} = rr^{*}$
  - zero-or-one instance of: $r? = r \mid \epsilon$
  - character classes: $[\mathbf{abd}] = \mathbf{a} \mid \mathbf{b} \mid \mathbf{d}$

    $[\mathbf{a} - \mathbf{z}] = \mathbf{a} \mid \mathbf{b} \mid \ldots \mid \mathbf{z}$

- Example, unsigned numbers:
  5280, 0.01234, 6.336E4, 1.89E-4

$$\begin{array}{rcl} \mathit{digit} & \rightarrow & [0 - 9] \\ \mathit{digits} & \rightarrow & \mathit{digit}^{+} \\ \mathit{number} & \rightarrow & \ldots \end{array}$$

Notational Shorthands

- We often use the following shorthands:
  - one-or-more instance of: $r^+ = rr^*$
  - zero-or-one instance of: $r? = r \mid \epsilon$
  - character classes: $[\mathbf{abd}] = \mathbf{a} \mid \mathbf{b} \mid \mathbf{d}$

    $[\mathbf{a} - \mathbf{z}] = \mathbf{a} \mid \mathbf{b} \mid \ldots \mid \mathbf{z}$

- Example, unsigned numbers:
  5280, 0.01234, 6.336E4, 1.89E-4

$$\begin{array}{rcl} \textit{digit} & \rightarrow & [0 - 9] \\ \textit{digits} & \rightarrow & \textit{digit}^+ \\ \textit{number} & \rightarrow & \textit{digits}(.\textit{digits})?(E[+-]?\textit{digits})? \end{array}$$

Notational Shorthands

- We often use the following shorthands:
  - one-or-more instance of: $r^+ = rr^*$
  - zero-or-one instance of: $r? = r \mid \epsilon$
  - character classes: $[\textbf{abd}] = \textbf{a} \mid \textbf{b} \mid \textbf{d}$

    $[\textbf{a} - \textbf{z}] = \textbf{a} \mid \textbf{b} \mid \ldots \mid \textbf{z}$

- Example, unsigned numbers:
  5280, 0.01234, 6.336E4, 1.89E-4

$$
\begin{array}{rcl}
\textit{digit} & \to & [0 - 9] \\
\textit{digits} & \to & \textit{digit}^+ \\
\textit{optionalFraction} & \to & .\textit{digits} \mid \epsilon \\
\textit{optionalExponent} & \to & (E(+ \mid - \mid \epsilon)digits) \mid \epsilon \\
\textit{number} & \to & \textit{digits optionalFraction optionalExponent}
\end{array}
$$

3.4 Recognition of Tokens

Grammar for branching statements:

$$\begin{array}{rcl} \textit{stmt} & \rightarrow & \textbf{if } \textit{expr } \textbf{then } \textit{stmt} \\ & | & \textbf{if } \textit{expr } \textbf{then } \textit{stmt } \textbf{else } \textit{stmt} \\ & | & \epsilon \\ \textit{expr} & \rightarrow & \textit{term } \textbf{relop } \textit{term} \\ & | & \textit{term} \\ \textit{term} & \rightarrow & \textbf{id} \\ & | & \textbf{number} \end{array}$$

Terminals are **if**, **then**, **else**, **relop**, **id** and **number**.
These are the names of the tokens.

Regular Definitions for Tokens

Regular definitions describing patterns for these tokens

$$
\begin{array}{rcl}
\textit{digit} & \to & [0-9] \\
\textit{digits} & \to & \textit{digit}^{+} \\
\textit{number} & \to & \textit{digits}(.\textit{digits})?(E[+-]?\textit{digits})? \\
\textit{letter} & \to & [\mathtt{A}-\mathtt{Z}\mathtt{a}-\mathtt{z}] \\
\textit{id} & \to & \textit{letter}(\textit{letter} \mid \textit{digit})^{*} \\
\textit{if} & \to & \mathtt{if} \\
\textit{then} & \to & \mathtt{then} \\
\textit{else} & \to & \mathtt{else} \\
\textit{relop} & \to & < \mid > \mid <= \mid >= \mid = \mid <>
\end{array}
$$

Regular definition for white space

$$\textit{ws} \to (\textbf{blank} \mid \textbf{tab} \mid \textbf{newline})^{+}$$

Lexemes and Their Tokens

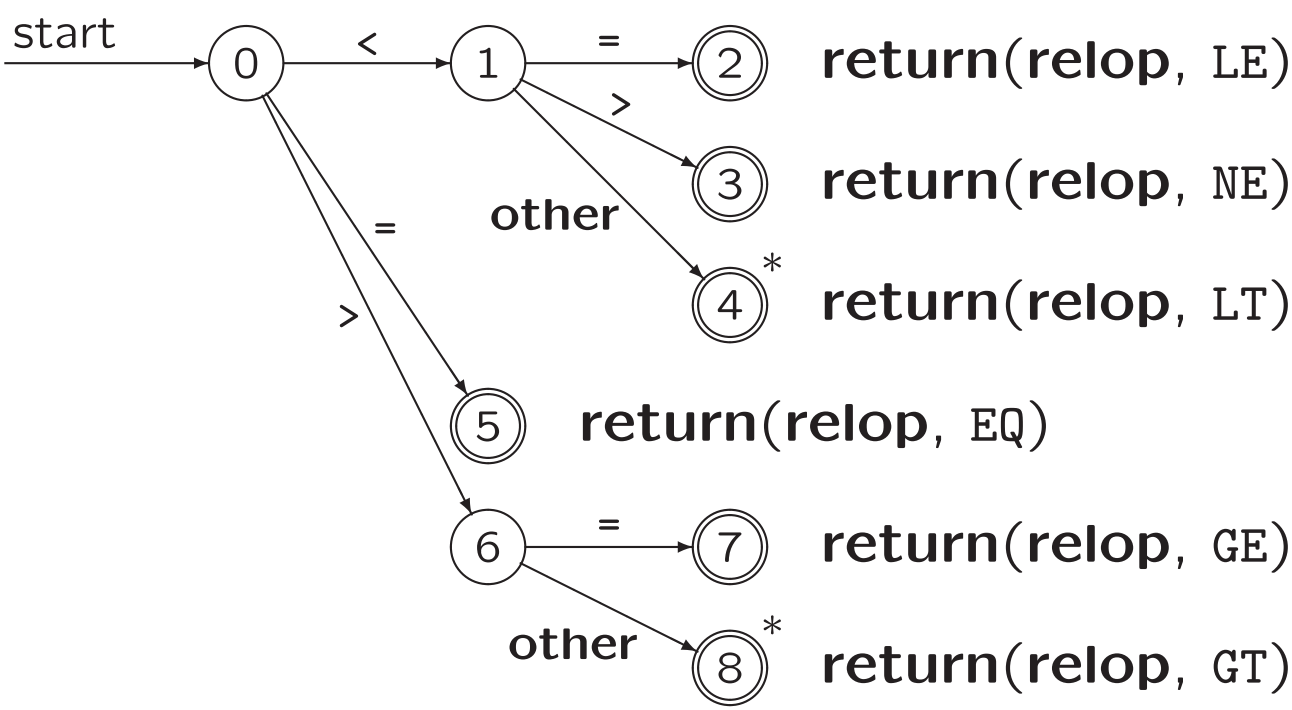
Goal:

Lexemes	Token name	Attribute value
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	pointer to table entry
Any *number*	**number**	pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

Transition Diagrams

(‘Almost finite automata’)

$$relop \rightarrow \; < \mid > \mid <= \mid >= \mid = \mid <>$$

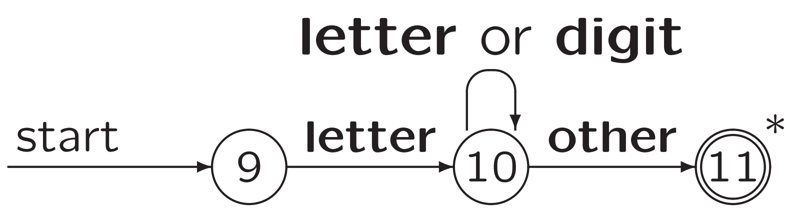


Retract input one position, if necessary ($*$)

Transition Diagrams

Identifiers and keywords

$$id \rightarrow letter(letter \mid digit)^*$$



return(*getToken*(), *installID*())

How to distinguish between identifiers and (reserved) keywords?

Transition Diagrams

How to distinguish between identifiers and (reserved) keywords?
Two possibilities:

- Install reserved words in symbol table initially
  Used in above diagram

- Separate transition diagram for each keyword
  Try these first, before the diagram for identifiers

From Diagram to Lexical Analyser

```
TOKEN getRelop ()
{ TOKEN retToken = new (RELOP);
  while (1)
  {  /* repeat character processing until a return
        or failure occurs */
    switch(state)
    { case 0: c = nextChar();
              if ( c == '<' ) state = 1;
              else if (c == '=' ) state = 5;
              else if (c == '>' ) state = 6;
              else fail(); /* lexeme is not a relop */
              break;
      case 1: ...
      ...
      case 8: retract();
              retToken.attribute = GT;
              return(retToken);
    }
  }
}
```

Entire Lexical Analyser

Based on transition diagrams for different tokens
How?

Entire Lexical Analyser

Based on transition diagrams for different tokens
Three possibilities:

- Try transition diagrams sequentially (in right order)
- Run transition diagrams in parallel
  Make sure to take longest prefix of input that matches any pattern
- Combine all transition diagrams into one

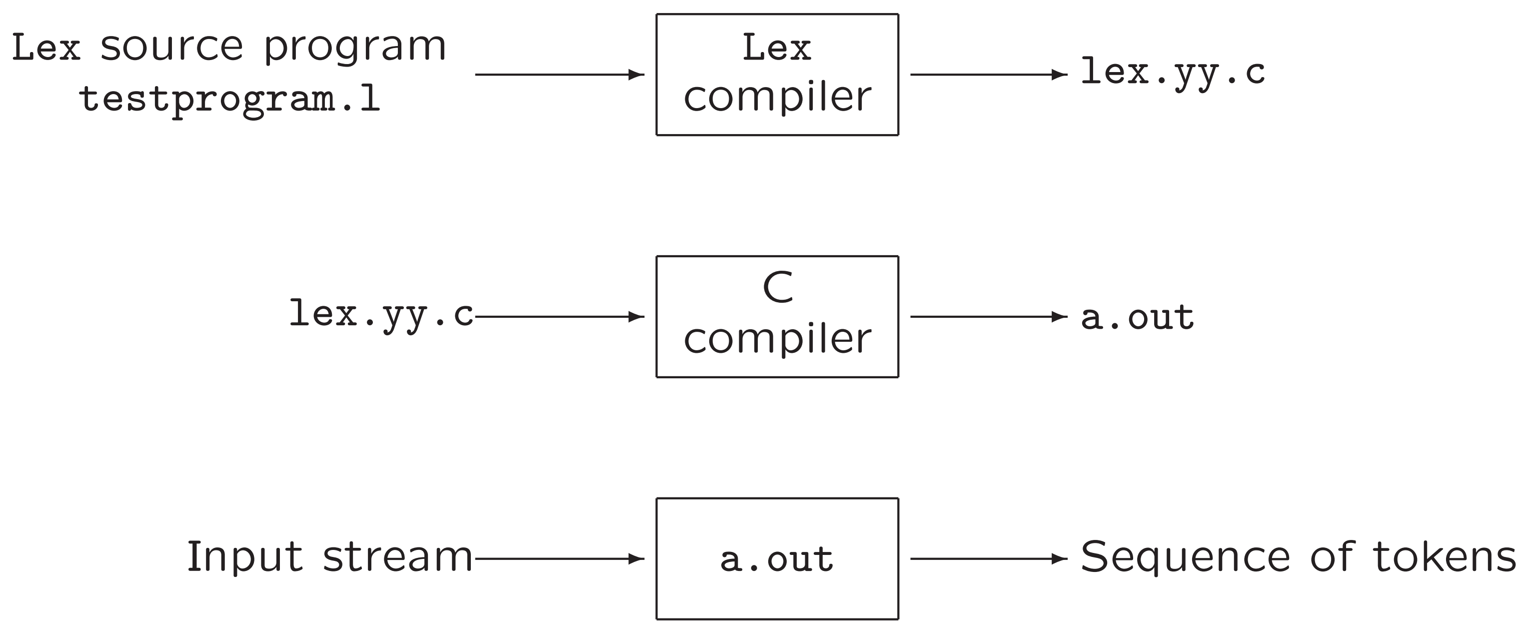
Implementing a Lexical Analyser

- By hand,
  using transition diagram to specify lexemes

Example of ‘ReadCommand’. . .

3.5 The Lexical-Analyser Generator Lex

Systematically translates regular definitions into C source code for efficient scanning



Structure of Lex Programs

- A `Lex` program has the following form

  declarations
  %%
  translation rules
  %%
  user defined auxiliary functions

- Translation rules are of the form

  Pattern   { Action }

  Patterns are `Lex` regular expressions

Operation of Lexical Analyser

The lexical analyser generated by `Lex`

- Activated by parser
- Reads input character by character
- Executes action A_i corresponding to pattern P_i
- Typically, A_i returns to the parser
- If not (e.g., in case of white space), proceed to find additional lexemes
- Lexical analyser returns single value: the token name
- Attribute value passed through global variable `yylval`

Regular Definitions for Tokens

Regular definitions describing patterns for these tokens

$$
\begin{array}{rcl}
\textit{digit} & \rightarrow & [0-9] \\
\textit{digits} & \rightarrow & \textit{digit}^{+} \\
\textit{number} & \rightarrow & \textit{digits}(.\textit{digits})?(E[+-]?\textit{digits})? \\
\textit{letter} & \rightarrow & [\texttt{A}-\texttt{Za}-\texttt{z}] \\
\textit{id} & \rightarrow & \textit{letter}(\textit{letter} \mid \textit{digit})^{*} \\
\textit{if} & \rightarrow & \texttt{if} \\
\textit{then} & \rightarrow & \texttt{then} \\
\textit{else} & \rightarrow & \texttt{else} \\
\textit{relop} & \rightarrow & < \mid > \mid <= \mid >= \mid = \mid <>
\end{array}
$$

Regular definition for white space

$$\textit{ws} \rightarrow (\textbf{blank} \mid \textbf{tab} \mid \textbf{newline})^{+}$$

Lexemes and Their Tokens

Goal:

Lexemes	Token name	Attribute value
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	pointer to table entry
Any *number*	**number**	pointer to table entry
`<`	**relop**	`LT`
`<=`	**relop**	`LE`
`=`	**relop**	`EQ`
`<>`	**relop**	`NE`
`>`	**relop**	`GT`
`>=`	**relop**	`GE`

The Lex Program (testprogram.l)

```
/* declarations section */
%{
    /* definitions of constants */
#define LT 256
    /* etcetera for LE, EQ, NE, GT, GE,
       IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions}
delim     [ \t\n]
ws        {delim}+
letter    [A-Za-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
number    {digit}+(\.{digit}+)?(E[+-]?{digit}+)?
```

The Lex Program (testprogram.l)

```
%%
/* translation rules section */

{ws}      {/* no action and no return */}
if        {return(IF);}
then      {return(THEN);}
else      {return(ELSE);}
{id}      {yylval = (int) installID(); return(ID);}
{number}  {yylval = (int) installNum(); return(NUMBER);}
"<"       {yylval = LT; return(RELOP);}
"<="      {yylval = LE; return(RELOP);}
"="       {yylval = EQ; return(RELOP);}
"<>"      {yylval = NE; return(RELOP);}
">"       {yylval = GT; return(RELOP);}
">="      {yylval = GE; return(RELOP);}

%%
/* auxiliary functions section */
int installID() {...}
int installNum() {...}
```

Regular expressions in Lex

Operator characters: `\ " . ^ $ [ ] * + ? { } | /`

Expression	Matches	Example
c	non-operator character c	`a`
$\backslash c$	operator charater c literally	`\*`
$"s"$	string s literally	`"**"`
.	any character but newline	`a.*b`
^	beginning of a line	`^abc`
$	end of a line	`abc$`
$[s]$	any one of the characters in string s	`[abc]`
$[\hat{\ }s]$	any one character not in string s	`[^abc]`
$[c_1 - c_2]$	any one character between c_1 and c_2	`[a-z]`
$r*$	zero or more strings matching r	`a*`
$r+$	one or more strings matching r	`a+`
$r?$	zero or one string matching r	`a?`
$r\{m,n\}$	between m and n occurrences of r	`a{1,5}`
$r_1 r_2$	an r_1 followed by an r_2	`ab`
$r_1 \mid r_2$	an r_1 or an r_2	`a|b`
(r)	same as r	`(a|b)`
r_1/r_2	r_1 when followed by r_2	`abc/123`
$\{d\}$	regular expression defined by d	`{id}`

Lex Details

- installID()
  function to install the lexeme into the symbol table
  returns pointer to symbol table entry
  `yytext` – pointer to the first character of the lexeme
  `yyleng` – length of the lexeme

- installNum()
  similar to installID, but puts numerical constants into a separate table

Lex Details

- Example: input "`\t\tif` "
  - Longest initial prefix: "`\t\t`" = *ws*
    No action, so `yytext` points to '`i`' and continue
  - Next lexeme is "`if`"
    Token **if** is returned, `yytext` points to '`i`' and `yyleng`=2
- Ambiguity and longest pattern matching:
  - Patterns `if` and {`id`} match lexeme "`if`"
  - If input is "`<=`", then lexeme is "`<=`"
- ```
  lex testprogram.l
  gcc lex.yy.c -ll
  ./a.out < input
  ```
  ```

3.6 – 3.9 From Regular Expressions to Lexical Analysers

Roughly speaking, see FI2

Not for exam

Compilerconstructie

college 2

Symbol Table / Lexical Analysis

Chapters for reading: 2.6, 2.7, 3.1–3.5, 4.3.1

Next week: also werkcollege