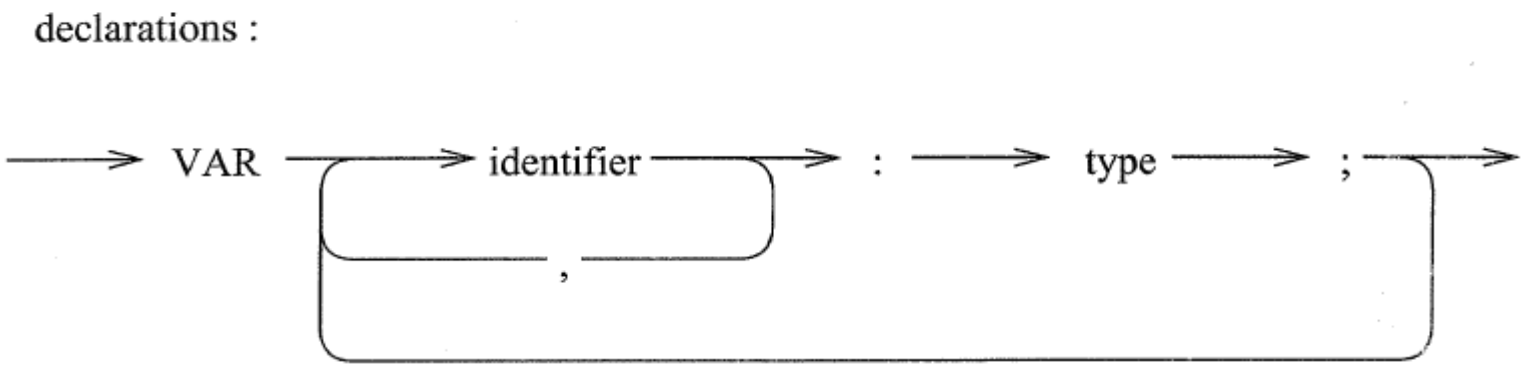
Preparing for the ACW

08348 Languages & Compilers

Introductory Lab

- There is an Introductory Lab
  - Just involves copying the lab task
  - See separate Lab slides

Language Roadmaps



- Convenient way of showing syntax in a diagram
- Used in ACW

Formal Specification of Languages

- Covered later in more detail
- Allows automation of language processing
- A language for describing languages
- A worthwhile investment of time
- Basic Structure

$$\alpha \rightarrow \beta$$

α, β are strings; collection of symbols

Symbols

- α, β are strings containing
  - Terminals
    - Language component
  - Non-terminals
    - Names of rules
- Example

  Sentence -> Noun Verb Noun

  Noun -> Bill

  Noun -> Jane

  Verb -> likes

  Verb -> knows

BNF

- Backus Naur Form
- Expressions enclosed in angle brackets
- Vertical line indicates OR (choice) operation
- Used to express language syntax rules

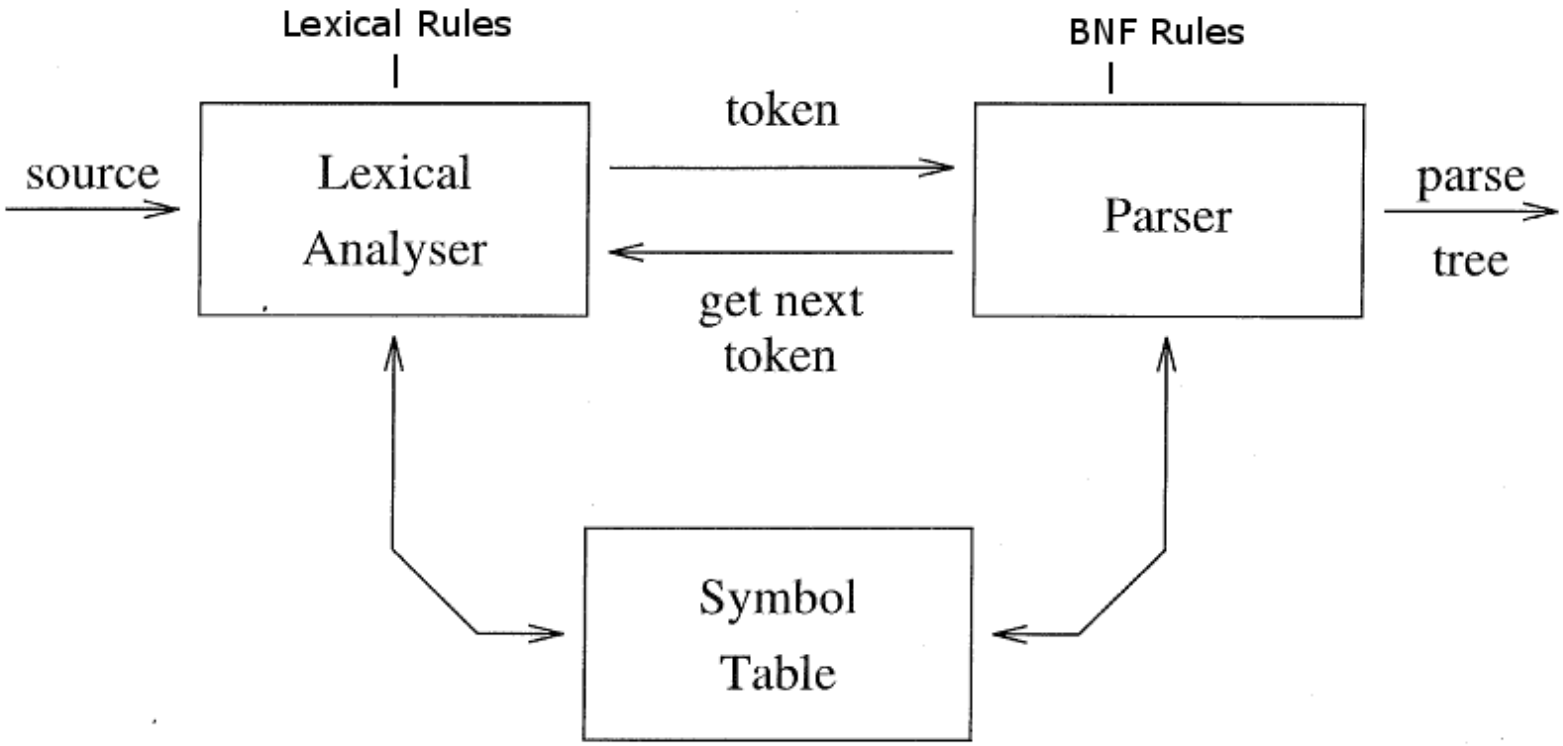
```
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<integer> ::= <digit> | <integer> <digit>
```

First ACW Task

- Convert Roadmaps to BNF
- Type this up, either in word or text file
  - Needed for creating parser later
- Do not delay or spend too long
  - Will delay coursework significantly
- Show to demonstrator to be signed off
- Do it at home before the lab
  - Bring question/problems to lab
- Save completed BNF for documentation later

Compiler Structure



Regular Expressions

- Covered in Discrete Maths
- Finite State Machines
- Used to describe elements of a language
  - Keywords : `BEGIN, END, FOR`
  - Numbers : `1  -2  3.74  2.6e5`
  - Variables : `count, a, Sum, Total`
  - Others: Comments/Strings

Writing Regular Expressions

- Define expressions using rules

```
Symbol = RegExp
```

```
Letter-A = a
Letter-B = “B”
Letter-C = ‘C’
Begin = BEGIN
```

Writing Regular Expressions

- Sets of Letters

```
digits = [0-9]
letters = [a-zA-Z]
```

- Names of Regular Expressions

```
cardinal = {number}
```

- Repetition
  - Use * for zero or more
  - Use + for 1 or more (note x+ == xx*)

Writing Regular Expression

- Use stick for choice
- Use parenthesis for grouping
- E.g:

```
letters = [a-zA-Z]
digits  = [0-9]
varname = {letters}({letters}|{digits})*
```

Second ACW Task

- Create `lex/flex` lexical analyser
- With regular expression for language elements
- Test on sample programs
- Save output for documentation later
- Show to demonstrator
- Start outside the lab
- Bring question problems to the lab
- Look at initial lab task for pointers

Lexical Analysis

- `lex` is a unix tool to build lexical analysers
  - `flex` is the same for linux/cygwin
  - Apple Mac has unix tools
- Define lexemes for tokens
- Generates program in C from rules
  - `lex.yy.c` C source for lexical analyser
  - `yylex()` function gives tokens

Lex Program

```
Declarations

%%

Translation rules

%%

Auxiliary procedures
```

Example Lex Program

```
delim    [ \t]
ws       {delim}+
digit    [0-9]
number   {digit}+
%%
          int k;
{ws}      ; /* white space, skip */
\n        printf("NEWLINE\n");
{number}  {k = atoi(yytext);
           printf("unsigned integer: %d\n",k);
           }
"+"       printf("binaryOp: PLUS\n");
"*"       printf("binaryOp: TIMES\n");
"("       printf("bracket: BRA\n");
")"       printf("bracket: KET\n");
```

Example Output

```
Unix % lex calcLex.l
Unix % wc -l lex.yy.c
     330 lex.yy.c
Unix % cc lex.yy.c -ll
Unix % ./a.out
    12    +    23
unsigned integer: 12
binaryOp: PLUS
unsigned integer: 23
NEWLINE
(1+2)*3
bracket: BRA
unsigned integer: 1
binaryOp: PLUS
unsigned integer: 2
bracket: KET
binaryOp: TIMES
unsigned integer: 3
NEWLINE
Unix %
```

```
Cygwin % flex calcLex.l


Cygwin % gcc lex.yy.c -o mylex.exe -lfl
Cygwin % ./mylex
```

```
^D              -- End of File
```

Lex syntax

- White Space has significance!
  - E.g. `word {letter} {digit}`
  - Means spaces required between letters and digits

- Translation Rules

  Expression Action

  Expression MUST be at start of line
  Action MUST NOT be at start of line

- Don’t leave lines of whitespace

Third ACW Task

- Create parser rules from BNF
  - Get `yacc/bison` to accept rules
    - Grammar ambiguities need resolving
  - Import tokens from lexical analyser
    - Need to modify lexer to return tokens
  - Test Parser rules using debug mode
  - Use example test programs
  - Save output for documentation
- Show to demonstrator

Building Parser

- Create Parser using `yacc`
  - Called `bison` on linux/cygwin
  - Available on Apple Mac
- Produces a parser in C
- Takes input from lexical analyser
- Language defined using grammar
  - (based on BNF)

YACC Program

```
Declarations

%%

Translation rules

%%

Auxiliary procedures
```

Calculator Example

- Consider Grammar

```
Lines  -> Line | Line Lines
Line   -> Expr newline
Expr   -> Expr + Term | Term
Term   -> Term * Factor | Factor
Factor -> ( Expr ) | Number
```

Calculator YACC

```
%token NUMBER PLUS TIMES BRA KET NEWLINE
%%
lines  : line
       | line lines
       ;
line   : expr NEWLINE          { printf ("line: value: %d\n", $1); }
       ;
expr   : expr PLUS term        {$$ = $1 + $3;}
       | term                  {$$ = $1;}
       ;
term   : term TIMES factor     {$$ = $1 * $3;}
       | factor                {$$ = $1;}
       ;
factor : BRA expr KET          {$$ = $2;}
       | NUMBER                {$$ = $1;}
       ;
%%
#include "lex.yy.c"
```

Calculator Lex Program

```
delim    [ \t]
ws       {delim}+
digit    [0-9]
number   {digit}+
%%
{ws}      ; /* white space, skip */
\n        return(NEWLINE);
{number} {yylval = atoi(yytext);
          return(NUMBER);
          }
"+"       return(PLUS);
"*"       return(TIMES);
"("       return(BRA);
")"       return(KET);
```

Building the Calculator

```
Unix % lex arithLex.l
Unix % yacc arithParse.y
Unix % cc y.tab.c -ly -ll
Unix % ./a.out
2 + (3 * 5)
Line: Value: 17
(1+5)
Line: value: 6

^D
Unix % ./a.out < Prog.a > Proga-out.txt
```

```
Unix % flex arithLex.l
Unix % bison arithParse.y
Unix % gcc arithParse.tab.c -lfl -o arith.exe
Unix % ./arith
```

Yacc Syntax

- Declaration part
  - Imports token from lex
  - Indicates start symbol
  - C language declarations
- Translation Rules
  - Of format

```
P : Rule { action}
```