Prolog

… the Parser

Review of Prolog 3: DCG

- Example grammar in DCG notation

/* rules with non-terminal symbols */

sentence → noun_phrase, verb_phrase.
noun_phrase → determiner, noun.
verb_phrase → verb.
verb_phrase → verb, noun_phrase.

/* rules with terminal symbols */

determiner → [the].
noun → [man].
noun → [apple].
verb → [sings].
verb → [eats].

Review of Prolog 3: DCG ➔ Prolog

Prolog converts the above DCG form to:-

```
// non-terminal symbols (NT)
sentence(A, C)        :- noun_phrase(A, B),  verb_phrase(B, C).
noun_phrase(A, C)     :- determiner(A, B),    noun(B, C).
verb_phrase(A, B)     :- verb(A, B).
verb_phrase(A, C)     :- verb(A, B),   noun_phrase(B, C).
// terminal symbols (T)
determiner( [the | A],  A).
noun( [man   |A], A).
noun( [apple |A], A).
verb(  [eats   |A], A).
verb(  [sings  |A], A).
```

Note how results are "**passed**"
Note for the terminals that this corresponds to **match** in the C parser. I.e. the input is a **list** and the **Terminal** is matched against the **head** of the list.
Success ➔ **tail** is "**returned**".

Prolog – call/exit/fail/redo

- These can be viewed as

- **CALL → EXIT** means a predicate has succeeded
- **CALL → FAIL** means a predicate has failed
- **REDO:** repeat until all possibilities have been found
- if more rules exist try these in turn until the process **FAILs**

- CALL / EXIT is “similar” to procedural programming
- REDO / FAIL is unique to Prolog

The Parser: start code

- **Parser: read source code + lexical analysis + syntax analysis**
- Prolog code: **Reader + Lexer + Parser**

```
ParseFile(File, Result) :-
    read_in(File, L), lexer(L, Tokens), parser(Tokens, Result).
```

file ➔ lexemes ➔ tokens

- This is turn may be packaged using tail recursion

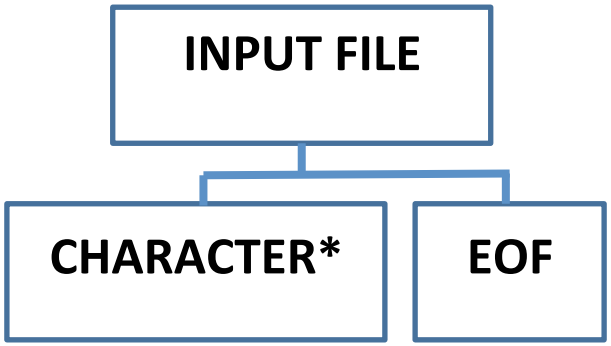
```
testa :- parseFiles(['testok1.pas', 'testok2.pas', 'testok3.pas']).

parseFiles([]).

parseFiles([H|T]) :- write('Testing '),  write(H), nl,
    read_in(H,L),  lexer(L, Tokens), parser(Tokens, _), nl,
    write(H), write(' end'), nl, nl, parseFiles(T).
```

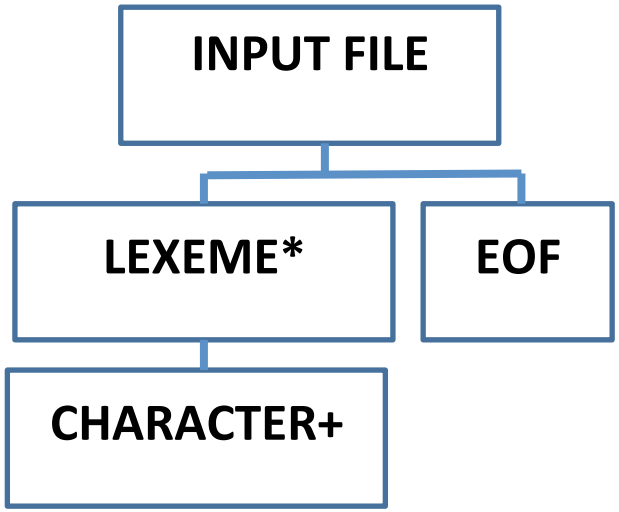
The Source File organisation

- Input File - **Physical**



* = zero or more
(think empty file!)

- Input File **Logical**



* = zero or more
(think empty file!)
+ = one or more

The simplified abstract view

- **The program text**

```
program testok1(input, output);
    var a, b, c: integer;
    begin
      a := b + c * 2
      end.
```

- **Program = list of lexemes**
  - Head = program
  - Tail = [testok1, (, input, ,, output,), ;, var, a, ,, b, ,, c, :, integer, ;, begin, a, :=, b, +, c, *, 2, end, .]
- **Lexeme = list of characters** e.g. [p, r, o, g, r, a, m]
  - Head = p
  - Tail = [r, o, g, r, a, m]

C parser versus Prolog parser

- C – (**Reader** + **Lexer**) + **Parser** **(Reader & Lexer combined)**
  - **Source file** ➔ (**Reader** + **Lexer**) ➔ **Tokens** ➔ **Parser**
- Prolog – **Reader** + **Lexer** + **Parser**
  - **Source file** ➔ **Reader** ➔ **Lexemes** ➔ **Lexer** ➔ **Tokens** ➔ **Parser**
- **Prolog Parser**
  - Uses **LISTs** to hold information – see the logical file organisation
  - **Reader**: source file ➔ **LIST** of lexemes
  - **Lexeme**: **LIST** of characters - i.e. a **string**
  - **Lexer**: **LIST** of lexemes ➔ **LIST** of tokens
  - **LIST**: **List ::= H T | ¤; H ::= element; T ::= List;**
- **C Parser**
  - Uses **arrays** – Lexeme buffer = **array** of characters – i.e. a **string**

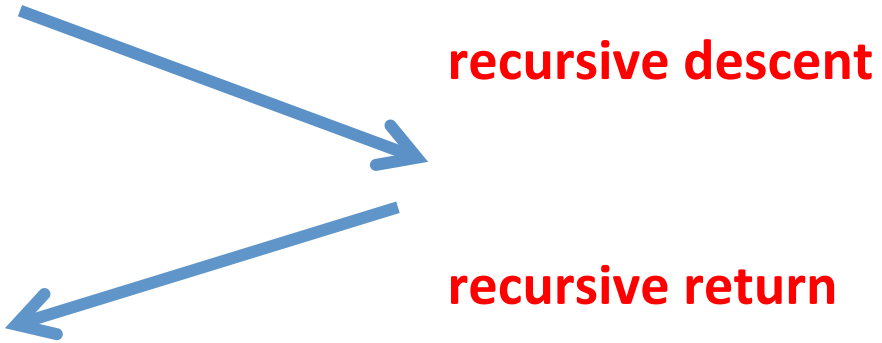
C parser versus Prolog parser

- **C Reader** – buffers the input file (string) & checks character by character (checks EOF)
- **C Lexer**
  1. White space removal
  2. Alphanumeric strings
  3. Numeric strings
  4. Special character (or string – one case “**:=**“)

- **Prolog Reader** – reads & checks character by character from the input file
- **Prolog Reader**
  1. Test for EOF
  2. Test for “**:**” or “**:=**“
  3. Test for single char
  4. Test for alphanumeric
  5. White space removal

Reading Prolog – nota bene!

- Be clear what the kind of an argument to a predicate is
  - Often character / list – look for signs of a list [H|T]
- Tail recursion is OFTEN used – look for tail recursion!
- In building character / word lists, the characters / words are often collected in the recursive DESCENT and assembled into words (i.e. a list of characters) / lists of atoms when returning from the recursive calls



The Reader

- This is an adapted form of the code in Clocksin & Mellish
- The file structure is
  - File is a sequence of "words" (lexemes)
  - Each "word" (lexeme) is a sequence of characters
  - The words and characters are collected and stored in lists
  - This is done using tail recursive functions
  - Use trace to see the details
- The start of the process is

```
read_in(File,[W|Ws]) :- see(File),  get0(C), // open file + get char
        readword(C, W, C1),      // read first word W
        restsent(W, C1, Ws),     // read rest of sentence
        nl, seen.                // nl = New Line, seen = close file
```

The Reader - mechanics

- **The start of the process is**

```
read_in(File,[W|Ws]) :- see(File),  get0(C), // open file + get char
        readword(C, W, C1),      // read first word W
        restsent(W, C1, Ws),     // read rest of sentence
        nl, seen.                // nl = New Line, seen = close file
```

- **Comments** - read_in(F, L) returns list of lexemes in L
  - File (physical) – a LIST of characters + EOF
  - File (logical) – a LIST of words (lexemes) + EOF
  - Mechanism – read the first word (lexeme) then the rest
  - Word (lexeme) – a LIST of characters
  - Mechanism – read the first character then the rest
  - get0(C) – read a character from a file (C get_char())

The Reader - mechanics

- **The start of the process is**

```
read_in(File,[W|Ws]) :- see(File),  get0(C), // open file + get char
        readword(C, W, C1),     // read first word W
        restsent(W, C1, Ws),    // read rest of sentence
        nl, seen.               // nl = New Line, seen = close file
```

- **read_in(File, [W|Ws])** - input: File output: lexeme list
- **see(File)** - open the input file
- **get0(C)** - read a character from the file
- **readword(C, W, C1)** - read a word (lexeme)
  - C = current character, W = lexeme, C1 next character after W
- **restsent(W, C1, Ws)** - read the rest of words
  - W = first word, C1 = next char after W, Ws = list of lexemes (words)

The Reader

readword(C+, W-, C1-) — **Tail Recursive**

- **C** – first char of the word , **W** – Word (lexeme)
- **C1/C2** – the first char **AFTER** the word W **(remember this!)**

```
readword(C, W, _)    :- C = -1, W = C.                /* EOF */
readword(C, W, C2)   :- C = 58, …                     /* ":" or ":=" */
readword(C, W, C1)   :- single_character(C),
                        name(W, [C]), get0(C1).
readword(C, W, C2)   :- in_word(C, NewC),             /* alpha & num */
                        get0(C1), restword(C1, Cs, C2),
                        name(W, [NewC|Cs]).
readword(_, W, C2)   :- get0(C1), readword(C1, W, C2).
```

+ = bound (instantiated) / - = unbound arguments (uninstantiated)

The Reader - explanation

readword(C, W, C1)

C = current char, W = word (lexeme),

C1 = next char after W

(this becomes the current char C in the next call)

Note That the readword definition is a sequence of tests

1. **Test for EOF**
2. **Test for ":" or ":="**
3. **Test for single character**
4. **Test for alpha & numeric lexemes (string)**
5. **Test for whitespace**

You may want to change this!

(αnum & number!)

```
readword(C, W, _)      :- C = -1, W = C.          /* EOF */
```

C = EOF, W = EOF, there is NO next char after W!

The Reader - explanation

The function name

name(W, [116, 103, 115, 116]). **gives W = test**

name(test, X). **gives X = [116, 103, 115, 116]**

i.e. name transforms a list of characters into an atom

OR

an atom into a list of characters

The Reader - explanation

```
readword(C, W, C2)     :- C = 58, …        /* ":" or ":="        */
                                           /* 58 is ':' 61 is '='  */
```

```
readword(C, W, C2) :- C = 58, get0(C1), readwordaux(C, W, C1, C2).
```

```
readwordaux(C, W, C1, C2) :- C1   = 61, name(W, [C, C1]), get0(C2).
readwordaux(C, W, C1, C2) :- C1 \= 61, name(W, [C]), C1 = C2.
```

name creates an atom from a list of characters – here either := or :

58 (':') has been found – examine the next character

(1) Next character 61 ('=') ➔ ":=" has been found

➔ get next character (C2)

(2) Next character NOT 61 ('=') ➔ ":" has been found

➔ return C1 as C2

The Reader - explanation

```
readword(C, W, C1)   :- single_character(C),
                        name(W, [C]), get0(C1).
```

```
single_character(40).      /* '(' */
single_character(41).      /* ')' */
single_character(42).      /* '*' */
single_character(43).      /* '+' */
single_character(44).      /* ',' */
single_character(46).      /* '.' */
single_character(58).      /* ':' */
single_character(59).      /* ';' */
```

The Reader - explanation

```
readword(C, W, C2)      :- in_word(C, NewC),    /* alpha & num */
                           get0(C1), restword(C1, Cs, C2),
                           name(W, [NewC|Cs]).

in_word(C, C) :- C > 96, C < 123.                  /* lower case */
in_word(C, L) :- C > 64, C < 91, L is C + 32.      /* upper case */
in_word(C, C) :- C > 47, C < 58.                   /* digits     */
```

in_word(C, NewC) C = character, NewC = return character

- lower case (97..122) ➔ NO CHANGE
- upper case (65..90) ➔ lower case
- digit (48..57) ➔ NO CHANGE

The Reader - explanation

```
readword(C, W, C2)      :- in_word(C, NewC),    /* alpha & num */
                           get0(C1),  restword(C1, Cs, C2),
                           name(W, [NewC|Cs]).
```

```
restword(C, [NewC|Cs], C2) :-
        in_word(C, NewC),
        get0(C1),
        restword(C1, Cs, C2).   /* tail recursive call          */

restword(C, [], C).             /* if C is not legal – in_word fails */
                                /* C belongs to the NEXT word    */
                                /* or is a separator 32 = space  */
```

e.g. C = p (word is program) then Cs = [r,o,g,r,a,m], (NB!) C2 = space
The recursive returns will give [], [m], [a,m], [r,a,m], … [r,o,g,r,a,m]

Reading:- “program¤” (¤=space)

readword(‘**p**’, W, C2) :- … get0(‘**r**’), **restword**(‘**r**’,Cs,C2), …
restword(‘**r**’, Cs, C2) :- … get0(‘**o**’), **restword**(‘**o**’,Cs,C2), …
restword(‘**o**’, Cs, C2) :- … get0(‘**g**’), **restword**(‘**g**’,Cs,C2), …
restword(‘**g**’, Cs, C2) :- … get0(‘**r**’), **restword**(‘**r**’,Cs,C2), …
restword(‘**r**’, Cs, C2) :- … get0(‘**a**’), **restword**(‘**a**’,Cs,C2), …
restword(‘**a**’, Cs, C2) :- … get0(‘**m**’), **restword**(‘**m**’,Cs,C2), …
restword(‘**m**’, Cs, C2) :- … get0(‘**¤**’), **restword**(‘**¤**’,Cs,C2),
…
restword(‘**¤**’, Cs, C2) :- **FAIL + RETRY** **restword**(‘**¤**’, [], ‘**¤**’) **STOP**

➔ **restword**(‘**m**’, [], ‘**¤**’) ➔ **restword**(‘**a**’, [m], ‘**¤**’)
➔ **restword**(‘**r**’, [a,m], ‘**¤**’) ➔ **restword**(‘**g**’, [r,a,m], ‘**¤**’)
➔ **restword**(‘**o**’, [g,r,a,m], ‘**¤**’) ➔ **restword**(‘**r**’, [o,g,r,a,m], ‘**¤**’)
➔ **readword**(‘**p**’, program, ‘**¤**’) W = program, C2 = ‘**¤**’

Reading:- “program¤” (¤=space)

Instantiation sequence **black** = instantiated **red** = to be instantiated

```
readword(C, W, C2)      :- in_word(C, NewC),    /* alphanum */
                           get0(C1),  restword(C1, Cs, C2),
                           name(W, [NewC|Cs]).
```

➔ C = ‘p’ ➔ in_word(‘p’, NewC) ➔ NewC = ‘p’ ➔ get0(C1) ➔ C1 = ‘r’
➔ restword(‘r’, Cs, C2) ➔ recursive calls (see previous slide)
➔ returns restword(‘r’, [o,g,r,a,m], ‘¤’)
➔ // Cs = [r,o,g,r,a,m], C2 = ‘¤’ ➔ readword(‘p’, W, ‘¤’)
➔ name(W, [‘p’ | [r,o,g,r,a,m]) ➔ W = program
➔ readword(‘p’, program, ‘¤’) ➔ W = program, C2 = ‘¤’

The Reader - explanation

readword(C, W, C2)

e.g. program space

```
:- in_word(C, NewC),    /* alpha & num */
   get0(C1),  restword(C1, Cs, C2),
   name(W, [NewC|Cs]).
```

readword(C, W, C2) C = p, W = undefined, C2 = undefined

in_word(C, NewC) C = p, NewC = p; get0(C1), C1 = r

restword (C1, Cs, C2) C1 = r, Cs = undefined, C2 = undefined

restword (C1, Cs, C2) C1 = o, Cs = undefined, C2 = undefined

etc. for program space – space is NOT alphanum!

in_word fails ➔ restword(space, [], space) Cs = [] empty list

recursive return [], [m], [a,m], …., Cs=[r,o,g,r,a,m], W = program

C2 = space ➔ readword(C, W, C2) C=p, W = program C2 = space

In the C reader we had lexbuff = “program”

```
if (isalpha(buffer[pbuf])) {
    while (isalnum(buffer[pbuf])) get_char();   etc  }
```

The Reader - explanation

readword(_, W, C2) **:- get0(C1), readword(C1, W, C2).**

(1) Not EOF

(2) Not 58 (':')

(3) Not a single character

(4) Not a word

(5) ➔ all other characters (e.g. space, tab, CR, LF) – IGNORE

if the character is none of EOF, ':', single character or in_word(C, L) (i.e. start building a new word)

then the character is ignored

The Reader - explanation

```
restsent(W, _, [])          :- W = -1.                    /* EOF stop */
restsent(W, _, [])          :- lastword(W).               /*  '.'  stop */
restsent(_, C, [W1|Ws])  :-
        readword(C, W1, C1),
        restsent(W1, C1, Ws).                  /* rest of sentence */
```

To test this try a trace on cmreader.txt & testok1.pas

?- read_in('cmreader.txt', L), write(L).

?- read_in('testok1.pas', L), write(L).

cmreader.txt ➔ [test, 55, :, .]

testok1.pas ➔ [program, testok1, (, input, ,, output,), ;,
var, a, ,, b, ,, c, :, integer, ;,
begin, a, :=, b, +, c, *, 2, end, .]

The Reader - explanation

```
restsent(W, _, [])          :- W = -1.                  /* EOF stop */
restsent(W, _, [])          :- lastword(W).             /*  '.'  stop */
restsent(_, C, [W1|Ws]) :-
        readword(C, W1, C1),
        restsent(W1, C1, Ws).       /* rest of sentence        */
```

Test: program testok1(input, output); … etc.

readword(C, W1, C1) gives C = p, W1 = program C1 = space

restsent(W1, C1, Ws) – readword skips the space & returns

W1 = testok1, C1 = (, Ws = [testok1]

restsent reads the lexemes Ws = [testok1, (, input, …, end, .]

read_in(File, [W|Ws]) then puts the W=program and Ws in a list – [program, testok1, (, input, …, end, .] i.e. the lexeme list

The Reader - mechanics

- **The start of the process is**

```
read_in(File,[W|Ws]) :- see(File),  get0(C), // open file + get char
        readword(C, W, C1),     // read first word W
        restsent(W, C1, Ws),    // read rest of sentence
        nl, seen.               // nl = New Line, seen = close file
```

- **read_in(File, [W|Ws])** - input: File output: lexeme list
- **see(File)** - open the input file
- **get0(C)** - read a character from the file
- **readword(C, W, C1)** - read a word (lexeme)
  - C = current character, W = lexeme, C1 next character after W
- **restsent(W, C1, Ws)** - read the rest of words
  - W = first word, C1 = next char after W, Ws = list of lexemes (words)

The Reader

readword(C+, W-, C1-) **Tail Recursive**

- **C** – first char of the word , **W** – Word (lexeme)
- **C1/C2** – the first char **AFTER** the word W **(remember this!)**

```
readword(C, W, _)    :- C = -1, W = C.                /* EOF */
readword(C, W, C2)   :- C = 58, …                     /* ":" or ":=" */
readword(C, W, C1)   :- single_character(C),
                        name(W, [C]), get0(C1).
readword(C, W, C2)   :- in_word(C, NewC),             /* alphanum */
                        get0(C1), restword(C1, Cs, C2),
                        name(W, [NewC|Cs]).
readword(_, W, C2)   :- get0(C1), readword(C1, W, C2).
```

+ = bound (instantiated) / - = unbound arguments (uninstantiated)

The simplified abstract view

- **The program text**

```
program testok1(input, output);
    var a, b, c: integer;
    begin
      a := b + c * 2
      end.
```

- **Program = list of lexemes**
  - Head = program
  - Tail = [testok1, (, input, ,, output,), ;, var, a, ,, b, ,, c, :, integer, ;, begin, a, :=, b, +, c, *, 2, end, .]
- **Lexeme = list of characters** e.g. [p, r, o, g, r, a, m]
  - Head = p
  - Tail = [r, o, g, r, a, m]

The Lexer

- The Reader produces a list of atoms (lexemes).
- The Lexer now has to transform these to tokens

```
read_in(File, L), lexer(L, Tokens), parser(Tokens, Result).
```

- The lexer becomes — note: tail recursive!

```
lexer([], []).
lexer([H|T], [F|S]) :- match(H, F), lexer(T, S).
```

and uses a predicate match(H, F) to transform a lexeme to a token

The Lexer

- **The predicate match(H, F) is:-**

```
match(L, T) :- L = 'program',     T is 256.
match(L, T) :- L = 'input',       T is 257.
match(L, T) :- L = 'output',      T is 258.
...
match(L, T) :- L = '(',           T is  40.
match(L, T) :- L = ')',           T is  41.
match(L, T) :- L = ',',           T is  44.
...
```

The Lexer

- **The predicate match(H, F) must also handle**
  - **Identifier**
  - **Number**
  - **Undefined**
  - **Error**

E.g. number

```
match(L, T) :- name(L, [H|Tail]), char_type(H, digit),
                    match_num(Tail), T is 272.
match_num([]).
match_num([H|T]) :- char_type(H, digit), match_num(T).
```

The Lexer

- **The lexer may then be tested using:-**

```
test_lexer(File, X) :- read_in(File, L), lexer(L, X), write(X).
```

- **Which for testok1.pas will give**

[256, 270, 40, 257, 44, 258, 41, 59, 259, 270, 44, 270, 44, 270, 58, 260, 59, 261, 270, 271, 270, 43, 270, 42, 272, 262, 46]

- **from**

[program, testok1, (, input, ,, output,), ;, var, a, ,, b, ,, c, :, integer, ;, begin, a, :=, b, +, c, *, 2, end, .]

The Parser

- **The Parser now checks the token list from the Lexer**
- **The Parser must define the grammar rules DCG + Terminals**
- **Keywords**

  program → [256].

  input → [257].
- **Id and number**

  id → [270].

  number → [272].
- **Symbols**

  lpar → [40].

  rpar → [41].

The Parser

- **The Parser grammar rules are:-**

```
prog          --> prog_header, var_part, stat_part.

prog_header --> program, id, lpar, input,
                comma, output, rpar, scolon.

...
```

- **Remember to remove left recursion!**
- **The parser is tested using**

```
read_in(H,L),  lexer(L, Tokens), parser(Tokens, _), nl,
```

Cross paradigm influences
The conditional expression in C - C / Prolog (logic)

```
static treeref b_rem(treeref T, int v)  {
return is_empty(T)              ? T
: v < get_value(node(T))        ? cons(b_rem(LC(T), v), node(T), RC(T))
: v > get_value(node(T))        ? cons(LC(T), node(T), b_rem(RC(T), v))
: is_empty(LC(T))               ? RC(T)              // 2 cases (¤, T, ¤)   (¤, T, RC)
: is_empty(RC(T))               ? LC(T)              // 1 case  (LC, T, ¤)
: HDiff(T) > 0                  ? LCmaxAsRoot(T)     // 1 case  (LC, T, RC)
:                                 RCminAsRoot(T);    // ¤ = empty
}
```

```
readword(C, W, C2) :-     C = 58, get0(C1), readwordaux(C, W, C1, C2).
readword(C, W, C1) :-     single_character(C), name(W, [C]), get0(C1).
readword(C, W, C2) :-     in_word(C, NewC),  get0(C1),
                          restword(C1, Cs, C2),  name(W, [NewC|Cs]).
readword(_, W, C2) :-     get0(C1), readword(C1, W, C2).
```

C and Prolog

- From this exercise, how can we compare C & Prolog?
- Look back at the Prolog code and try to identify patterns or programming "clichés"

- if (X) then (S1) else if … else if … else (Sn);

```
readword(C, W, _)     :- C = -1, W = C.          /* EOF */
readword(C, W, C2)    :- C = 58, …               /* ":=" */
readword(C, W, C1)    :- single_character(C),
                         name(W, [C]), get0(C1).
readword(C, W, C2)    :- in_word(C, NewC),    /* alpha & num */
                         get0(C1),  restword(C1, Cs, C2),
                         name(W, [NewC|Cs]).
readword(_, W, C2)    :- get0(C1),  readword(C1, W, C2).
```

C and Prolog

- From this exercise, how can we compare C & Prolog?
- Look back at the Prolog code and try to identify patterns or programming "clichés"

- if (X) then (S1) else (S2);

```
readwordaux(C, W, C1, C2) :- C1   = 61, name(W, [C, C1]), get0(C2).
readwordaux(C, W, C1, C2) :- C1 \= 61, name(W, [C]), C1 = C2.

parser(TList, Res) :- (prog(TList, Res), Res = [], write('Parse OK!'));
                      write('Parse Fail!').
```

C and Prolog

- From this exercise, how can we compare C & Prolog?
- Look back at the Prolog code and try to identify patterns or programming "clichés"

- case / switch

```
match(L, T) :- L = 'program',     T is 256.
match(L, T) :- L = 'input',       T is 257.
match(L, T) :- L = 'output',      T is 258.
```

The default flow of control (foc) is the sequence

Repetition is realised via RECURSION

Reader, Lexer, Parser

Summary

- **Results are "passed" from one predicate to another**

```
read_in(File, L), lexer(L, Tokens), parser(Tokens, Result).
```

- **Check how results are "passed" back or forward**
- **Tail recursion is the most common form of repetition**

```
lexer([], []).
lexer([H|T], [F|S]) :- match(H, F), lexer(T, S).
```

- **Check the arguments to predicates: atoms or lists [H|T]**
- **Read each predicate separately and UNDERSTAND how it works – use trace to help see the result**
- **Identify programming clichés & relate them to what you know (if-then-else; if … else if … else; switch;)**