

CS 460

Programming Languages

Fall 2008

Dr. Watts

(Week 9)



Prolog Factorial Example

```
GNU Prolog 1.3.0
By Daniel Diaz
Copyright (C) 1999-2007 Daniel Diaz
| ?- ['factorial.pl'].
compiling
/home/faculty/tiawatts/cs460f08/factorial.pl
for byte code...
/home/faculty/tiawatts/cs460f08/factorial.pl
compiled, 7 lines read - 905 bytes written, 5
ms
yes
| ?- listing.
factorial(0, 1).
factorial(A, B) :-
    A > 0,
    C is A - 1,
    factorial(C, D),
    B is A * D.
yes
| ?- factorial(1,R).
R = 1 ?
yes
| ?- factorial(2,R).
R = 2 ?
yes
| ?- factorial(3,R).
R = 6 ?
yes
```

```
| ?- trace.
The debugger will first creep -- showing
everything (trace)
yes
{trace}
| ?- factorial(2,R).
1 1 Call: factorial(2,_16) ?
2 2 Call: 2>0 ?
2 2 Exit: 2>0 ?
3 2 Call: _113 is 2-1 ?
3 2 Exit: 1 is 2-1 ?
4 2 Call: factorial(1,_138) ?
5 3 Call: 1>0 ?
5 3 Exit: 1>0 ?
6 3 Call: _190 is 1-1 ?
6 3 Exit: 0 is 1-1 ?
7 3 Call: factorial(0,_215) ?
7 3 Exit: factorial(0,1) ?
8 3 Call: _243 is 1*1 ?
8 3 Exit: 1 is 1*1 ?
4 2 Exit: factorial(1,1) ?
9 2 Call: _16 is 2*1 ?
9 2 Exit: 2 is 2*1 ?
1 1 Exit: factorial(2,2) ?
R = 2 ?
(1 ms) yes
{trace}
| ?- notrace.
The debugger is switched off
yes
| ?-
```



list.pl



```
is_first(Element,[Element|_]).

is_last(Element,[Element]).
is_last(Element,[_|Tail]) :- is_last(Element,Tail).

is_in(Element,[Element|_]).
is_in(Element,[_|Tail]) :- is_in(Element,Tail).

how_long(0,[]).
how_long(1,[_]).
how_long(N,[_|Tail]) :- how_long(M,Tail),
                        N is M+1.

place(Element,[Element|_],1).
place(Element,[_|Tail],N) :- place(Element,Tail,M),
                              M > 0,
                              N is M+1.

insert_first(Element,List,[Element|List]).

remove_first([],[]).
remove_first(_|Tail,Tail).

concat([],List,List).
concat([Element|List1],List2,[Element|List3]) :- concat(List1,List2,List3).
```

First and Follow Sets



- **Firsts**
 - A terminal symbol T_i is a member of the First Set of non-terminal symbol $\langle nt_j \rangle$ if T_i can become the first terminal symbol in a complete expansion of $\langle nt_j \rangle$.
- **Follows**
 - A terminal symbol T_i is a member of the Follow Set of non-terminal symbol $\langle nt_j \rangle$ if T_i can become the first terminal symbol immediately following a complete expansion of $\langle nt_j \rangle$.

Calculating First and Follow Sets – Rule 1



1. $\langle \text{program} \rangle \rightarrow \text{begin } \langle \text{stmt_list} \rangle \text{ end}$
 2. $\langle \text{stmt_list} \rangle \rightarrow \langle \text{stmt} \rangle \langle \text{stmt_tail} \rangle$
 3. $\langle \text{stmt_tail} \rangle \rightarrow ; \langle \text{stmt_list} \rangle$
 4. $\langle \text{stmt_tail} \rangle \rightarrow \lambda$
 5. $\langle \text{stmt} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expression} \rangle$
 6. $\langle \text{var} \rangle \rightarrow \mathbf{A}$
 7. $\langle \text{var} \rangle \rightarrow \mathbf{B}$
 8. $\langle \text{var} \rangle \rightarrow \mathbf{C}$
 9. $\langle \text{expression} \rangle \rightarrow \langle \text{var} \rangle \langle \text{expr_tail} \rangle$
 10. $\langle \text{expr_tail} \rangle \rightarrow + \langle \text{var} \rangle \langle \text{expr_tail} \rangle$
 11. $\langle \text{expr_tail} \rangle \rightarrow * \langle \text{var} \rangle \langle \text{expr_tail} \rangle$
 12. $\langle \text{expr_tail} \rangle \rightarrow \lambda$
1. For each rule of the form
 $\langle \text{nt}_i \rangle \rightarrow T_k \dots$
 T_k is included in the first set of $\langle \text{nt}_i \rangle$

Calculating First and Follow Sets – Rule 2



1. $\langle \text{program} \rangle \rightarrow \text{begin } \langle \text{stmt_list} \rangle \text{ end}$
 2. $\langle \text{stmt_list} \rangle \rightarrow \langle \text{stmt} \rangle \langle \text{stmt_tail} \rangle$
 3. $\langle \text{stmt_tail} \rangle \rightarrow ; \langle \text{stmt_list} \rangle$
 4. $\langle \text{stmt_tail} \rangle \rightarrow \lambda$
 5. $\langle \text{stmt} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expression} \rangle$
 6. $\langle \text{var} \rangle \rightarrow \mathbf{A}$
 7. $\langle \text{var} \rangle \rightarrow \mathbf{B}$
 8. $\langle \text{var} \rangle \rightarrow \mathbf{C}$
 9. $\langle \text{expression} \rangle \rightarrow \langle \text{var} \rangle \langle \text{expr_tail} \rangle$
 10. $\langle \text{expr_tail} \rangle \rightarrow + \langle \text{var} \rangle \langle \text{expr_tail} \rangle$
 11. $\langle \text{expr_tail} \rangle \rightarrow * \langle \text{var} \rangle \langle \text{expr_tail} \rangle$
 12. $\langle \text{expr_tail} \rangle \rightarrow \lambda$
2. For each rule of the form
 $\langle \text{nt}_i \rangle \rightarrow \langle \text{nt}_j \rangle \dots$
if T_k is a member of the first set of $\langle \text{nt}_j \rangle$ then T_k is included in the first set of $\langle \text{nt}_i \rangle$

Calculating First and Follow Sets – Rule 3



1. $\langle \text{program} \rangle \rightarrow \text{begin } \langle \text{stmt_list} \rangle \text{ end}$
 2. $\langle \text{stmt_list} \rangle \rightarrow \langle \text{stmt} \rangle \langle \text{stmt_tail} \rangle$
 3. $\langle \text{stmt_tail} \rangle \rightarrow ; \langle \text{stmt_list} \rangle$
 4. $\langle \text{stmt_tail} \rangle \rightarrow \lambda$
 5. $\langle \text{stmt} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expression} \rangle$
 6. $\langle \text{var} \rangle \rightarrow \mathbf{A}$
 7. $\langle \text{var} \rangle \rightarrow \mathbf{B}$
 8. $\langle \text{var} \rangle \rightarrow \mathbf{C}$
 9. $\langle \text{expression} \rangle \rightarrow \langle \text{var} \rangle \langle \text{expr_tail} \rangle$
 10. $\langle \text{expr_tail} \rangle \rightarrow + \langle \text{var} \rangle \langle \text{expr_tail} \rangle$
 11. $\langle \text{expr_tail} \rangle \rightarrow * \langle \text{var} \rangle \langle \text{expr_tail} \rangle$
 12. $\langle \text{expr_tail} \rangle \rightarrow \lambda$
3. For each rule of the form $\langle \text{nt}_i \rangle \rightarrow \lambda$ if T_k is a member of the follow set of $\langle \text{nt}_i \rangle$ then T_k is included in the first set of $\langle \text{nt}_i \rangle$

Calculating First and Follow Sets – Rule 4



1. $\langle \text{program} \rangle \rightarrow \text{begin } \langle \text{stmt_list} \rangle \text{ end}$
 2. $\langle \text{stmt_list} \rangle \rightarrow \langle \text{stmt} \rangle \langle \text{stmt_tail} \rangle$
 3. $\langle \text{stmt_tail} \rangle \rightarrow ; \langle \text{stmt_list} \rangle$
 4. $\langle \text{stmt_tail} \rangle \rightarrow \lambda$
 5. $\langle \text{stmt} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expression} \rangle$
 6. $\langle \text{var} \rangle \rightarrow \mathbf{A}$
 7. $\langle \text{var} \rangle \rightarrow \mathbf{B}$
 8. $\langle \text{var} \rangle \rightarrow \mathbf{C}$
 9. $\langle \text{expression} \rangle \rightarrow \langle \text{var} \rangle \langle \text{expr_tail} \rangle$
 10. $\langle \text{expr_tail} \rangle \rightarrow + \langle \text{var} \rangle \langle \text{expr_tail} \rangle$
 11. $\langle \text{expr_tail} \rangle \rightarrow * \langle \text{var} \rangle \langle \text{expr_tail} \rangle$
 12. $\langle \text{expr_tail} \rangle \rightarrow \lambda$
4. For each rule of the form $\langle \text{nt}_i \rangle \rightarrow \dots \langle \text{nt}_j \rangle T_k \dots$ T_k is included in the follow set of $\langle \text{nt}_j \rangle$

Calculating First and Follow Sets – Rule 5



1. $\langle \text{program} \rangle \rightarrow \mathbf{begin} \langle \text{stmt_list} \rangle \mathbf{end}$
 2. $\langle \text{stmt_list} \rangle \rightarrow \langle \text{stmt} \rangle \langle \text{stmt_tail} \rangle$
 3. $\langle \text{stmt_tail} \rangle \rightarrow ; \langle \text{stmt_list} \rangle$
 4. $\langle \text{stmt_tail} \rangle \rightarrow \lambda$
 5. $\langle \text{stmt} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expression} \rangle$
 6. $\langle \text{var} \rangle \rightarrow \mathbf{A}$
 7. $\langle \text{var} \rangle \rightarrow \mathbf{B}$
 8. $\langle \text{var} \rangle \rightarrow \mathbf{C}$
 9. $\langle \text{expression} \rangle \rightarrow \langle \text{var} \rangle \langle \text{expr_tail} \rangle$
 10. $\langle \text{expr_tail} \rangle \rightarrow + \langle \text{var} \rangle \langle \text{expr_tail} \rangle$
 11. $\langle \text{expr_tail} \rangle \rightarrow * \langle \text{var} \rangle \langle \text{expr_tail} \rangle$
 12. $\langle \text{expr_tail} \rangle \rightarrow \lambda$
5. For each rule of the form
 $\langle \rangle \rightarrow \dots \langle \text{nt}_i \rangle \langle \text{nt}_j \rangle \dots$
if T_k is a member of the first set of $\langle \text{nt}_i \rangle$ then T_k is included in the follow set of $\langle \text{nt}_j \rangle$

Calculating First and Follow Sets – Rule 6



1. $\langle \text{program} \rangle \rightarrow \mathbf{begin} \langle \text{stmt_list} \rangle \mathbf{end}$
 2. $\langle \text{stmt_list} \rangle \rightarrow \langle \text{stmt} \rangle \langle \text{stmt_tail} \rangle$
 3. $\langle \text{stmt_tail} \rangle \rightarrow ; \langle \text{stmt_list} \rangle$
 4. $\langle \text{stmt_tail} \rangle \rightarrow \lambda$
 5. $\langle \text{stmt} \rangle \rightarrow \langle \text{var} \rangle = \langle \text{expression} \rangle$
 6. $\langle \text{var} \rangle \rightarrow \mathbf{A}$
 7. $\langle \text{var} \rangle \rightarrow \mathbf{B}$
 8. $\langle \text{var} \rangle \rightarrow \mathbf{C}$
 9. $\langle \text{expression} \rangle \rightarrow \langle \text{var} \rangle \langle \text{expr_tail} \rangle$
 10. $\langle \text{expr_tail} \rangle \rightarrow + \langle \text{var} \rangle \langle \text{expr_tail} \rangle$
 11. $\langle \text{expr_tail} \rangle \rightarrow * \langle \text{var} \rangle \langle \text{expr_tail} \rangle$
 12. $\langle \text{expr_tail} \rangle \rightarrow \lambda$
6. For each rule of the form
 $\langle \text{nt}_i \rangle \rightarrow \dots \langle \text{nt}_j \rangle$
if T_k is a member of the follow set of $\langle \text{nt}_i \rangle$ then T_k is included in the follow set of $\langle \text{nt}_j \rangle$

Parse Table for Grammar for Small Programming Language



	begin	end	;	=	A	B	C	+	*
program	1								
stmt_list					2	2	2		
stmt					5	5	5		
stmt_tail		4	3						
var					6	7	8		
expression					9	9	9		
expr_tail		12	12					10	11

```

int program()
{
    df << "Starting <program>. Current token = " << names[token] << endl;
    int errors = 0;
    int firsts [] = {BEGIN_TOK, EOF_TOK};
    int follows [] = {EOF_TOK};
    if (token == BEGIN_TOK)
    { // Rule 1
        GetToken ();
        errors += stmt_list ();
        if (token == END_TOK)
            GetToken ();
        }
        else
        {
            errors++;
            error_msg ("'" + words[END_TOK] + "' expected");
        }
        if (token == EOF_TOK)
            GetToken ();
        }
        else
        {
            errors++;
            error_msg ("'" + words[EOF_TOK] + "' expected");
        }
    }
    else
    {
        errors++;
        error_msg ("unexpected '" + lexeme + "'");
        find_in_ffset (1, follows);
    }
    df << "Ending <program>. Current token = " << names[token] << ". Errors = " << errors << endl;
    return errors;
}

```



```

int stmt_list()
{
    df << "Starting <stmt_list>. Current token = " << names[token]
    << endl;
    int errors = 0;
    int firsts [] = {A_TOK, B_TOK, C_TOK, EOF_TOK};
    int follows [] = {END_TOK, EOF_TOK};
    if (!InSet(token, 3, firsts))
    {
        errors++;
        error_msg ("unexpected '" + lexeme + "'");
        find_in_ffset (4, firsts, 2, follows);
    }
    if (token == A_TOK || token == B_TOK || token == C_TOK)
    { // Rule 2
        errors += stmt ();
        errors += stmt_tail ();
    }
    else
    {
        errors++;
        error_msg ("unexpected '" + lexeme + "'");
        find_in_ffset (2, follows);
    }
    df << "Ending <stmt_list>. Current token = " << names[token] <<
    ". Errors = " << errors << endl;
    return errors;
}

```



```

int stmt_tail()
{
    df << "Starting <stmt_tail>. Current token = " << names[token] << endl;
    int errors = 0;
    int firsts [] = {SEMI_TOK, END_TOK, EOF_TOK};
    int follows [] = {END_TOK, EOF_TOK};
    if (!InSet(token, 2, firsts))
    {
        errors++;
        error_msg ("unexpected '" + lexeme + "'");
        find_in_ffset (3, firsts, 2, follows);
    }
    if (token == SEMI_TOK)
    { // Rule 3
        GetToken ();
        errors += stmt_list ();
    }
    else if (token == END_TOK)
    { // Rule 4
        ;
    }
    else
    {
        errors++;
        error_msg ("unexpected '" + lexeme + "'");
        find_in_ffset (2, follows);
    }
    df << "Ending <stmt_tail>. Current token = " << names[token] << ". Errors = "
    << errors << endl;
    return errors;
}

```



```

int stmt()
{
    df << "Starting <stmt>. Current token = " << names[token] << endl;
    int errors = 0;
    int firsts [] = {A_TOK, B_TOK, C_TOK, EOF_TOK};
    int follows [] = {END_TOK, SEMI_TOK, EOF_TOK};
    if (!InSet(token, 3, firsts))
    {
        errors++;
        error_msg ("unexpected '" + lexeme + "'");
        find_in_ffset (4, firsts, 3, follows);
    }
    if (token == A_TOK || token == B_TOK || token == C_TOK)
    { // Rule 5
        errors += var ();
        if (token == EQUAL_TOK)
            GetToken ();
        }
        else
        {
            errors++;
            error_msg ("'" + words[EQUAL_TOK] + "' expected");
        }
        errors += expr ();
    }
    else
    {
        errors++;
        error_msg ("unexpected '" + lexeme + "'");
        find_in_ffset (3, follows);
    }
    df << "Ending <stmt>. Current token = " << names[token] << ". Errors = " << errors << endl;
    return errors;
}

```



```

int var()
{
    df << "Starting <var>. Current token = " << names[token] << endl;
    int errors = 0;
    int firsts [] = {A_TOK, B_TOK, C_TOK, EOF_TOK};
    int follows [] = {END_TOK, SEMI_TOK, EQUAL_TOK, PLUS_TOK, MULT_TOK, EOF_TOK};
    if (!InSet(token, 3, firsts))
    {
        errors++;
        error_msg ("unexpected '" + lexeme + "'");
        find_in_ffset (4, firsts, 6, follows);
    }
    if (token == A_TOK)
    { // Rule 6
        GetToken ();
    }
    else if (token == B_TOK)
    { // Rule 7
        GetToken ();
    }
    else if (token == C_TOK)
    { // Rule 8
        GetToken ();
    }
    else
    {
        errors++;
        error_msg ("unexpected '" + lexeme + "'");
        find_in_ffset (6, follows);
    }
    df << "Ending <var>. Current token = " << names[token] << ". Errors = " << errors << endl;
    return errors;
}

```



```

int expr()
{
    df << "Starting <expr>. Current token = " << names[token] << endl;
    int errors = 0;
    int firsts [] = {A_TOK, B_TOK, C_TOK, EOF_TOK};
    int follows [] = {END_TOK, SEMI_TOK, EOF_TOK};
    if (!InSet(token, 3, firsts))
    {
        errors++;
        error_msg ("unexpected '" + lexeme + "'");
        find_in_ffset (4, firsts, 3, follows);
    }
    if (token == A_TOK || token == B_TOK || token == C_TOK)
    { // Rule 9
        errors += var ();
        errors += expr_tail ();
    }
    else
    {
        errors++;
        error_msg ("unexpected '" + lexeme + "'");
        find_in_ffset (3, follows);
    }
    df << "Ending <expr>. Current token = " << names[token] << ". Errors = "
    << errors << endl;
    return errors;
}

```



```

int expr_tail()
{
    df << "Starting <expr_tail>. Current token = " << names[token] << endl;
    int errors = 0;
    int firsts [] = {PLUS_TOK, MULT_TOK, END_TOK, SEMI_TOK, EOF_TOK};
    int follows [] = {END_TOK, SEMI_TOK, EOF_TOK};
    if (!InSet(token, 4, firsts))
    {
        errors++;
        error_msg ("unexpected '" + lexeme + "'");
        find_in_ffset (5, firsts, 3, follows);
    }
    if (token == PLUS_TOK)
    { // Rule 10
        GetToken ();
        errors += var ();
    }
    else if (token == MULT_TOK)
    { // Rule 11
        GetToken ();
        errors += var ();
    }
    else if (token == END_TOK || token == SEMI_TOK)
    { // Rule 12
        ;
    }
    else
    {
        errors++;
        error_msg ("unexpected '" + lexeme + "'");
        find_in_ffset (3, follows);
    }
    df << "Ending <expr_tail>. Current token = " << names[token] << ". Errors = " << errors << endl;
    return errors;
}

```





Data Type

- Collection of data values and a set of predefined operations on those values.
- User-defined - COBOL
- Abstract data types – Smalltalk - ALGOL
- Descriptor
 - Collection of the attributes of the variable
 - Amount and format of the memory associated with a variable



Primitive Data Types

- Numeric Types
- Integer
 - Byte, short, int, long, long long
 - Unsigned vs. signed
 - Binary vs. twos compliment
- Floating Point
 - IEEE format
 - Sign bit, exponent, fraction
 - Precision vs. range – trade off
 - Non-terminating values (eg. 0.1)

IEEE Single Precision



- The IEEE single precision floating point standard representation requires a 32 bit word, which may be represented as numbered from 0 to 31, left to right. The first bit is the sign bit, S, the next eight bits are the exponent bits, 'E', and the final 23 bits are the fraction 'F':

```
S EEEEEEEE FFFFFFFFFFFFFFFFFFFFFFFF
0 1      8 9                          31
```

- The value V represented by the word may be determined as follows:
 - If E=255 and F is nonzero, then V=NaN ("Not a number")
 - If E=255 and F is zero and S is 1, then V=-Infinity
 - If E=255 and F is zero and S is 0, then V=Infinity
 - If 0<E<255 then $V=(-1)^S * 2^{(E-127)} * (1.F)$ where "1.F" is intended to represent the binary number created by prefixing F with an implicit leading 1 and a binary point.
 - If E=0 and F is nonzero, then $V=(-1)^S * 2^{(E-126)} * (0.F)$ These are "unnormalized" values.
 - If E=0 and F is zero and S is 1, then V=-0
 - If E=0 and F is zero and S is 0, then V=0
- In particular,
 - 0 00000000 000000000000000000000000 = 0
 - 1 00000000 000000000000000000000000 = -0
 - 0 11111111 000000000000000000000000 = Infinity
 - 1 11111111 000000000000000000000000 = -Infinity
 - 0 11111111 000001000000000000000000 = NaN
 - 1 11111111 00100010001001010101010 = NaN
 - 0 10000000 000000000000000000000000 = $+1 * 2^{(128-127)} * 1.0 = 2$
 - 0 10000001 101000000000000000000000 = $+1 * 2^{(129-127)} * 1.101 = 6.5$
 - 1 10000001 101000000000000000000000 = $-1 * 2^{(129-127)} * 1.101 = -6.5$
 - 0 00000001 000000000000000000000000 = $+1 * 2^{(1-127)} * 1.0 = 2^{(-126)}$
 - 0 00000000 100000000000000000000000 = $+1 * 2^{(-126)} * 0.1 = 2^{(-127)}$
 - 0 00000000 000000000000000000000001 = $+1 * 2^{(-126)} * 0.000000000000000000000001 = 2^{(-149)}$ (Smallest positive value)

IEEE Double Precision



- The IEEE double precision floating point standard representation requires a 64 bit word, which may be represented as numbered from 0 to 63, left to right. The first bit is the sign bit, S, the next eleven bits are the exponent bits, 'E', and the final 52 bits are the fraction 'F':

```
S EEEEEEEEEEE FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
0 1      11 12                                     63
```

- The value V represented by the word may be determined as follows:
 - If E=2047 and F is nonzero, then V=NaN ("Not a number")
 - If E=2047 and F is zero and S is 1, then V=-Infinity
 - If E=2047 and F is zero and S is 0, then V=Infinity
 - If 0<E<2047 then $V=(-1)^S * 2^{(E-1023)} * (1.F)$ where "1.F" is intended to represent the binary number created by prefixing F with an implicit leading 1 and a binary point.
 - If E=0 and F is nonzero, then $V=(-1)^S * 2^{(E-1022)} * (0.F)$ These are "unnormalized" values.
 - If E=0 and F is zero and S is 1, then V=-0
 - If E=0 and F is zero and S is 0, then V=0