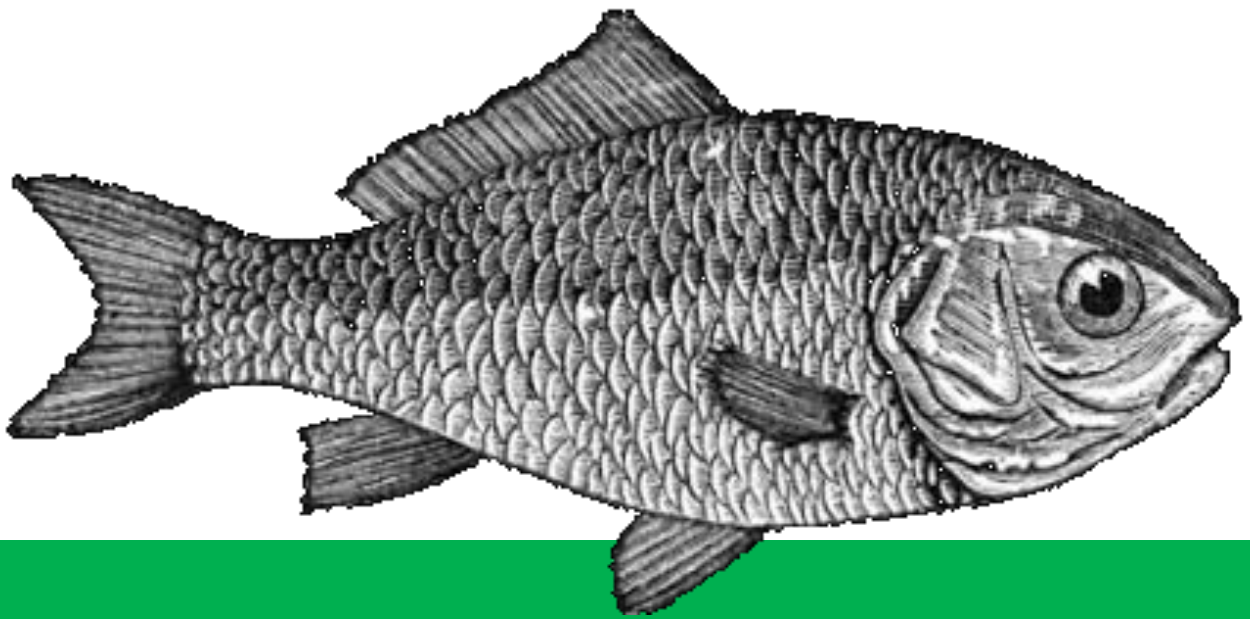
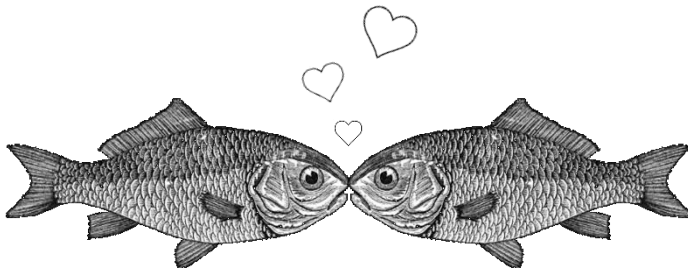


*Primitive Guide*

**Version 1.2**  
Updated on 22-02-2017



A Fifth-generation Programming Language

# PROLOG

Very high level language for beginner

FREE FOR ALL

*Willow Fung*  
2017.



# Basic

Command line> `awk -f awkprolog.awk parser.pro proc_1`

## Facts and Rules

File: father.pro	Output
father(johnny, thomas). father(samuel, david). father(james, kevin). father(thomas, edward). father(david, leo). father(david, peter). father(david, gordan). father(edward, joe).  grandfather(X, Z) :- father(X, Y), father(Y, Z).  ?- grandfather(Who, peter), writeln(Who).  ?- grandfather(samuel, Who), writeln(Who), printmore.	samuel yes  leo yes  leo peter gordan fail

File: parent.pro	Output
human(david). human(john). human(suzie). human(eliza). man(david). man(john). woman(suzie). woman(eliza). parent(david, john). parent(john, eliza). parent(suzie, eliza).  father(X,Y) :- parent(X,Y), man(X). mother(X,Y) :- parent(X,Y), woman(X).  ?- father(Who, eliza), writeln(Who).	john yes  suzie eliza yes



# Arithmetic

## Factorial

File: fac.pro	Output
<pre>fac(1, 1) :- !. fac(N, F) :- N2 is N - 1, fac(N2, F2), F is F2 * N.  ?- fac(5, R), writeln(R).</pre>	<pre>120 yes</pre>

## Fibonacci

File: fib.pro	Output
<pre>fib(0, 0). fib(1, 1). fib(X, Y) :-     X &gt; 1,     X2 is X - 2,     fib(X2, Y2),     X1 is X - 1,     fib(X1, Y1),     Y is Y1 + Y2.  ?- fib(7, F), writeln(F).</pre>	<pre>13 yes</pre>



# List Operations

File: member.pro	Output
<pre>member(X, [X _]). member(X, [_ L]) :- member(X, L).  ?- member(b,[a,b,c]). ?- member(X,[a,b,c]), writeln(X), allresult.</pre>	<pre>yes  a b c fail</pre>

File: append.pro	Output
<pre>append([], L, L). append([H T], L, [H L2]) :- append(T, L, L2).  ?- append([a, b], [c, d], C), writeln(C). ?- append(A, B, [a, b, c, d]),    write(A), write(', '), writeln(B), all.</pre>	<pre>[a, b, c, d] yes  [], [a, b, c, d] [a], [b, c, d] [a, b], [c, d] [a, b, c], [d] [a, b, c, d], [] Fail</pre>

File: length.pro	Output
<pre>length([], 0). length([_ L], N) :- length(L, N1), N is N1 + 1.  ?- length([a, b, c, d, e], LEN), writeln(LEN).</pre>	<pre>5 yes</pre>

File: length.pro	Output
<pre>last(X, [X]). last(X, [_ L]) :- last(X, L).  ?- last(X, [a, b, c, d, e]), writeln(X).</pre>	<pre>e yes</pre>

File: element_n.pro	Output
<pre>at(X, [X _], 1). at(X, [_ L], N) :- N &gt; 1, N1 is N - 1, at(X,L,N1).  ?- at(X, [a, b, c, d, e], 3), writeln(X).</pre>	<pre>c yes</pre>

File: reverse.pro	Output
<pre> rev(L1, L2) :- rev2(L1, L2, []).  rev2([],L,L). rev2([X Xs], L2, Acc) :- rev2(Xs, L2, [X Acc]).  ?- rev([a, b, c, d, e], L), writeln(L).</pre>	<pre> [e, d, c, b, a] yes</pre>

File: drop.pro	Output
<pre> drop(X, [X Xs], 1, Xs). drop(X, [Y Xs], K, [Y Ys]) :- K &gt; 1,     K1 is K - 1, drop(X, Xs, K1, Ys).  ?- N is 3, drop(X, [a,b,c,d,e], N, R), writeln(X), writeln(R).</pre>	<pre> c [a, b, d, e] yes</pre>

File: compress.pro	Output
<pre> compress([], []). compress([X], [X]). compress([X,X Xs], Zs) :- compress([X Xs], Zs). compress([X,Y Ys], [X Zs]) :- notequal(X, Y),     compress([Y Ys], Zs).  notequal(X, X) :- !, fail. notequal(X, Y).  ?- compress([a,a,a,a,b,c,c,a,a,d,d,e,e,e,e], X), writeln(X).</pre>	<pre> [a, b, c, a, d, e] yes</pre>

File: pack.pro	Output
<pre> pack([], []). pack([X Xs], [Z Zs]) :- transfer(X, Xs, Ys, Z),     pack(Ys, Zs).  transfer(X, [], [], [X]). transfer(X, [Y Ys], [Y Ys], [X]) :- notequal(X, Y). transfer(X, [X Xs], Ys, [X Zs]) :-     transfer(X, Xs, Ys, Zs).  notequal(X, X) :- !, fail. notequal(X, Y).  ?- pack([a,a,a,a,b,c,c,a,a,d,e,e,e,e], X), writeln(X).</pre>	<pre> [a, b, c, a, d, e] yes</pre>

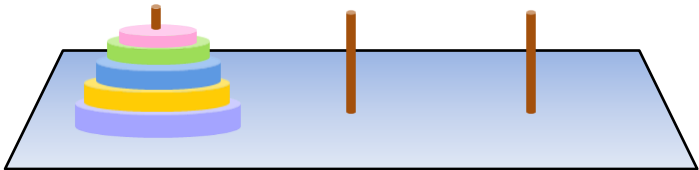


# Problem Solving

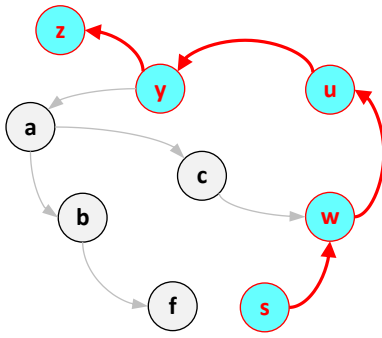
## Run-length encoding

File: encoding.pro	Output
<pre>notequal(X, X) :- !, fail. notequal(X, Y).  length([], 0). length(_ L, N) :- length(L, N1), N is N1 + 1.  pack([], []). pack([X Xs], [Z Zs]) :- transfer(X, Xs, Ys, Z), pack(Ys, Zs).  transfer(X, [], [], [X]). transfer(X, [Y Ys], [Y Ys], [X]) :- notequal(X, Y). transfer(X, [X Xs], Ys, [X Zs]) :- transfer(X, Xs, Ys, Zs).  transform([], []). transform([X Xs] Ys, [[N,X] Zs]) :- length([X Xs], N), transform(Ys, Zs).  encode(L1, L2) :- pack(L1, L), transform(L, L2).  ?- encode([a,a,b,c,c,c,a,a,d,e,e,e,e], X), writeln(X).</pre>	<pre>[[2, a], [1, b], [3, c], [2, a], [1, d], [4, e]] yes</pre>

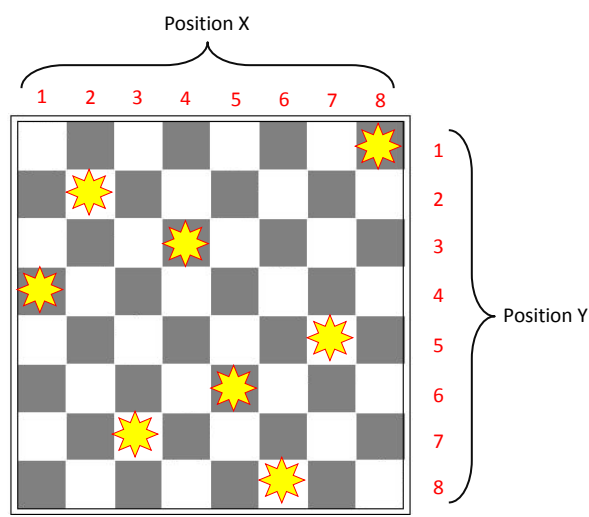
## Hanoi Tower

File: hanoi.pro	Output
<pre> hanoi(N) :- move(N, left, middle, right). move(1, A, U, C) :- inform(A, C), !. move(N, A, B, C) :-     N1 is N - 1,     move(N1, A, C, B),     inform(A, C),     move(N1, B, A, C).  inform(P1, P2) :- write(P1), write(' -&gt; '),     writeln(P2).  ?- hanoi(5). </pre> 	<pre> left -&gt; right left -&gt; middle right -&gt; middle left -&gt; right middle -&gt; left middle -&gt; right left -&gt; right left -&gt; middle right -&gt; middle right -&gt; left middle -&gt; left right -&gt; middle left -&gt; right left -&gt; middle right -&gt; middle left -&gt; right middle -&gt; left middle -&gt; right left -&gt; right left -&gt; middle right -&gt; middle left -&gt; right middle -&gt; left middle -&gt; right left -&gt; right yes </pre>

## Find Path

File: path.pro	Output
<pre> a(a, c). a(a, b). a(b, f). a(s, w). a(y, a). a(y, z). a(w, u). a(u, y). a(c, w).  not(G) :- G, !, fail. not(G).  path(X, X, T, [X]). path(X, Z, T, [X T2]) :-     a(X, Y),     not(member(Y, T)),     path(Y, Z, [Y T], T2).  member(X, [X _]). member(X, [_ T]) :- member(X, T).  ?- path(s, z, [], P),     write('The path from s to z : '),     writeln(P). </pre>	<p>The path from s to z : [s, w, u, y, z] yes</p> 

## Eight Queen Problem

File: queen8.pro	Output
<pre> member(X, [X _]). member(X, [_ T]) :- member(X, T).  sol([]). sol([X / Y   Other]) :-     sol(Other),     member(Y, [1,2,3,4,5,6,7,8]),     noattack(X / Y, Other).  noattack(_, []). noattack(X / Y, [X1 / Y1   Other]) :-     Y \= Y1,     Y1 - Y \= X1 - X,     Y1 - Y \= X - X1,     noattack(X/Y, Other).  template([1/Y1, 2/Y2, 3/Y3, 4/Y4,           5/Y5, 6/Y6, 7/Y7, 8/Y8]).  ?- template(S), sol(S), writeln(S). </pre>	<p>[pos(1, 4), pos(2, 2), pos(3, 7), pos(4, 3), pos(5, 6), pos(6, 8), pos(7, 5), pos(8, 1)] yes</p> 



## Four Color Map

File: map4color.pro

Output

```
color(red).
color(green).
color(blue).
color(yellow).

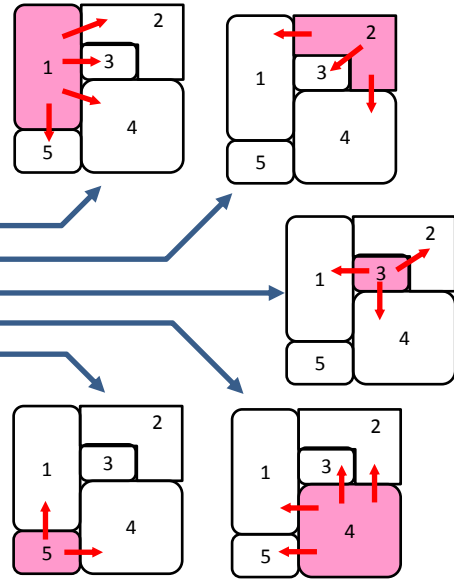
n(StateAColor, StateBColor) :- color(StateAColor),
    color(StateBColor),
    not(eq(StateAColor, StateBColor)).

not(G) :- G, !, fail.
not(G).
eq(X, X).
```

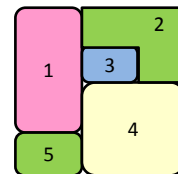
```
*****
% Define states in a Country
*****
```

```
country(S1, S2, S3, S4, S5) :-
    n(S1, S2), n(S1, S3), n(S1, S4), n(S1, S5),
    n(S2, S1), n(S2, S3), n(S2, S4),
    n(S3, S1), n(S3, S2), n(S3, S4),
    n(S4, S1), n(S4, S2), n(S4, S3), n(S4, S5),
    n(S5, S1), n(S5, S4).
```

```
?- R is country(S1, S2, S3, S4, S5),
    R,
    writeln(R).
```



country(red, green, blue, yellow, green)  
yes





# Basic Sorting Algorithm

File: sortbubble.pro	Output
<pre>sort(L, S) :- swap(L, L1), !, sort(L1, S). sort(S, S). swap([X, Y   R], [Y, X   R]) :- X &gt; Y. swap([Z   R], [Z   R1]) :- swap(R, R1).  ?- sort([6, 45, 30, 21, 23, 20, 8, 7, 32, 1], S), writeln(S).</pre>	<pre>[1, 6, 7, 8, 20, 21, 23, 30, 32, 45] yes</pre>

File: sortquick.pro	Output
<pre>qsort([], []). qsort([X T], S) :-     split(X, T, SM, BI),     qsort(SM, SSM),     qsort(BI, SBI),     conc(SSM, [X SBI], S).  split(X, [], [], []). split(X, [Y T], [Y S], B) :- X &gt; Y, !, split(X, T, S, B). split(X, [Y T], S, [Y B]) :- split(X, T, S, B).  conc([], L, L). conc([X L1], L2, [X L3]) :- conc(L1, L2, L3).  ?- qsort([6, 45, 30, 21, 23, 20, 8, 7, 32, 1], S), writeln(S).</pre>	<pre>[1, 6, 7, 8, 20, 21, 23, 30, 32, 45] yes</pre>



# Computer Language Prototype

## File: parser.pro

```
eat(err, L, L) :- !.
eat(X, [X|T], T) :- !.
true.

%*****
block(Va, Vb, Vc) :-
    eat(begin, Vb, Vd), actions(Va, Vd, Ve), eat(end, Ve, Vc).
actions([Va|Vb], Vc, Vd) :-
    action(Va, Vc, Ve), more_actions(Vb, Ve, Vd).
more_actions(Va, Vb, Vc) :- eat(sep, Vb, Vd), actions(Va, Vd, Vc).
more_actions([], Va, Va) :- true.
action(if(Va, Vb, Vc), Vd, Ve) :-
    eat(if, Vd, Vf), cond(Va, Vf, Vg), eat(then, Vg, Vh), block(Vb, Vh, Vi),
    eat(else, Vi, Vj), block(Vc, Vj, Vd).
action(while(Va, Vb), Vc, Vd) :-
    eat(while, Vc, Ve), cond(Va, Ve, Vf), eat(do, Vf, Vg), block(Vb, Vg, Vd).
action(assign(Va, Vb), Vc, Vd) :-
    eat(id(Va), Vc, Ve), eat(eq, Ve, Vf), expr(Vb, Vf, Vd).
expr(Va, Vb, Vc) :-
    term(Vd, Vb, Ve), exprtail(Vd, Va, Ve, Vc).
exprtail(Va, add(Va, Vb), Vc, Vd) :-
    eat(add, Vc, Ve), term(Vf, Ve, Vg), exprtail(Vf, Vb, Vg, Vd).
exprtail(Va, sub(Va, Vb), Vc, Vd) :-
    eat(sub, Vc, Ve), term(Vf, Ve, Vg), exprtail(Vf, Vb, Vg, Vd).
exprtail(Va, mul(Va, Vb), Vc, Vd) :-
    eat(mul, Vc, Ve), term(Vf, Ve, Vg), exprtail(Vf, Vb, Vg, Vd).
exprtail(Va, div(Va, Vb), Vc, Vd) :-
    eat(div, Vc, Ve), term(Vf, Ve, Vg), exprtail(Vf, Vb, Vg, Vd).
exprtail(Va, Va, Vb, Vb) :- true.
term(id(Va), Vb, Vc) :- eat(id(Va), Vb, Vc).
term(const(Va), Vb, Vc) :- eat(const(Va), Vb, Vc).

cond(Va, Vb, Vc) :- eat(id(Vd), Vb, Ve), ctail(Vd, Va, Ve, Vc).
ctail(Va, eq(Va, Vb), Vc, Vd) :- eat(eq, Vc, Ve), expr(Vb, Ve, Vd).
ctail(Va, ne(Va, Vb), Vc, Vd) :- eat(ne, Vc, Ve), expr(Vb, Ve, Vd).
ctail(Va, idtest(Va), Vb, Vb) :- true.
```

## File: proc\_1.pas

```
?- L = [begin, id("X"), eq, const(5), add, const(3), sep, id("Y"), eq, id("X"), sub, const(2),
end], block(T, L, R), writeln(T).
```

```
?- L = [if, id("X"), eq, const(3), add, id("Y"), then, begin, id("X"), eq, const(3), end,
else, begin, id("X"), eq, const(4), end], actions(T, L, R), writeln(T).
```

Command line> **awk -f awkprolog.awk parser.pro proc\_1.pas**

### Original code:

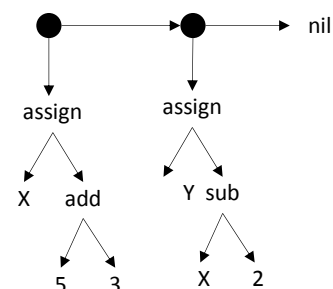
```
begin
  X = 5 + 3 ;
  Y = X - 2 ;
end
```

### Prolog list format:

```
[begin,
 id("X"), eq, const(5), add, const(3), sep,
 id("Y"), eq, id("X"), sub, const(2),
end]
```

### Parse to Abstract Syntax Tree:

```
[ assign(X, add(const(5), const(3))),
  assign(Y, sub(id(X), const(2))) ]
```



**File: compiler.pro**

```
cg(I, [pushc(I)]) :- number(I).
cg(A, [push(A)]) :- atomic(A).

cg(add(X, Y), [CX, CY, add]) :- cg(X, CX), cg(Y, CY).
cg(sub(X, Y), [CX, CY, sub]) :- cg(X, CX), cg(Y, CY).
cg(mul(X, Y), [CX, CY, mul]) :- cg(X, CX), cg(Y, CY).

cg(cmp(s_gt, X, Y), [CX, CY, gt]) :- cg(X, CX), cg(Y, CY).
cg(cmp(s_lt, X, Y), [CX, CY, lt]) :- cg(X, CX), cg(Y, CY).
cg(cmp(s_equ, X, Y), [CX, CY, equ]) :- cg(X, CX), cg(Y, CY).
cg(cmp(not_equ, X, Y), [CX, CY, not_equ]) :- cg(X, CX), cg(Y, CY).
cg(cmp(s_ge, X, Y), [CX, CY, ge]) :- cg(X, CX), cg(Y, CY).
cg(cmp(s_le, X, Y), [CX, CY, le]) :- cg(X, CX), cg(Y, CY).

cg(func(main, X, Y), [CY, halt]) :- cg(Y, CY).
cg(func(F, X, Y), [pushc(R1), pop(F), bz(R2), label(R1), CY, ret, label(R2)]) :- cg(Y, CY).

cg(assign_stm(X, Y), [CY, pop(X)]) :- cg(Y, CY).
cg(while_stm(X, S), [label(R1), CX, bz(R2), SX, br(R1), label(R2)]) :-
    cg(X, CX), cg(S, SX).

cg([], []).
cg([A|B], [CA, CB]) :- cg(A, CA), cg(B, CB).

append([], L, L).
append([H|T], L, [H|L2]) :- append(T, L, L2).

flatten([], []).
flatten([H|T], L3) :- flatten(H, L1), flatten(T, L2), append(L1, L2, L3).
flatten(X, [X]).

find_symbol([], D, D) :- !.
find_symbol([push(X) | T], D, D3) :- add_sym(X, D, D2), find_symbol(T, D2, D3).
find_symbol([pop(X) | T], D, D3) :- add_sym(X, D, D2), find_symbol(T, D2, D3).
find_symbol([H | T], D, DD) :- find_symbol(T, D, DD).

member(X, [X | L]) :- !.
member(X, [_ | L]) :- member(X, L).

add_sym(X, L, L) :- member(X, L).
add_sym(X, L, [X | L]).

allocate([], N, []).
allocate([H | T], N, [sym(H, N) | T2]) :- N2 is N + 1, allocate(T, N2, T2).

relocate([], SYM, []).
relocate([push(X) | T], D, [push(Y) | T2]) :-
    member(sym(X, Y), D), relocate(T, D, T2).
relocate([pop(X) | T], D, [pop(Y) | T2]) :-
    member(sym(X, Y), D), relocate(T, D, T2).
relocate([H | T], D, [H | T2]) :- relocate(T, D, T2).

out_code([], N) :- !.
out_code([label(A)|T], N) :- !, out_code(T, N).
out_code([H|T], N) :- !, write(N), write(':'), writeln(H), N2 is N + 1, out_code(T, N2).

ass([], N, N, D, D) :- !.
ass([label(A)|T], A, B, D, D2) :- !, ass(T, A, B, D, D2).
ass([H|T], N, NN, D, D2) :- !, N2 is N + 1, ass(T, N2, NN, D, D2).

gen(T) :- writeln('compile...'), cg(T, C), flatten(C, L),
    writeln('assemble...'), ass(L, 0, HEAP, [], FUN_LIST),
    writeln('generate symbol table...'), find_symbol(L, [], SYM_LIST),
    writeln('allocate...'), allocate(SYM_LIST, HEAP, SYM_ADDR),
    writeln('relocate...'), relocate(L, SYM_ADDR, L2),
    out_code(L2, 0),
    writeln(SYM_ADDR),
    writeln(FUN_LIST).
```

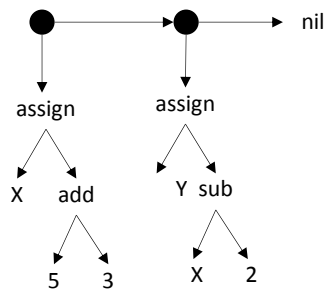
**File: proc\_2.ast**

```
?- AST = func(main, void,  
    [assign_stm(a, 10),  
      assign_stm(b, add(a, 2))]  
) , gen(AST).
```

Command line> **awk -f awkprolog.awk compiler.pro proc\_2.ast**

**Parse to Abstract Syntax Tree:**

[ assign(x, add(5, 3)),  
 assign(y, sub(x, 2)) ]

**Intermediate Instruction Generation:**

```
ast  
compile...  
assemble...  
generate symbol table...  
allocate...  
relocate...  
0:pushc(5)  
1:pushc(3)  
2:add  
3:pop(10)  
4:push(10)  
5:pushc(2)  
6:sub  
7:pop(9)  
8:halt  
[sym(y, 9), sym(x, 10)]  
[]  
yes
```

**Memory Slot Allocation:**

Addr	Instruction	Description
0	pushc 5	Push const 5
1	pushc 3	Push const 3
2	add	Calculate (add) and return result to stack
3	pop 10	Pop a value(8) from stack to address 10 (var X's addr.)
4	push 10	Push from address 10 (var X's addr.)
5	pushc 2	Push const 2
6	sub	Calculate (subtract) and return result to stack
7	pop 9	Pop a value(7) from stack to address 9 (var Y's addr.)
8	halt	Stop te program
9	Y	Address of Variable Y
10	X	Address of Variable X

**Symbol Table:**

Symbol	Address
Y	9
X	10