```
% Load in utilities for reading tokens and
% outputting the tree representation of the program
:- [util].

%% Boolean Expressions consists of
%% an expression followed by a comparison
%% operator followed by another expression

comparison(comp(E1,CO,E2)) --> expression(E1),cop(CO),expression(E2).

cop(eq) --> ['=='].
cop(le) --> ['<'].
cop(ge) --> ['>'].
cop(leq) --> ['<='].
cop(geq) --> ['>='].
cop(neq) --> ['!='].

%% Arithmetic Expressions -- more complicated to
%% ensure that precedence of operators are handled
%% correctly and that we do not get into an
%% infinite loop

expression(E) --> term(E).
expression(E) --> term(T), exprP(T,E).

exprP(T,expr(O,T,T2)) --> wop(O), term(T2).
exprP(T,E) --> wop(O), term(T2), exprP(expr(O,T,T2),E).

term(T) --> factor(T).
term(T) --> factor(F), termP(F,T).

termP(F,expr(O,F,F2),T) --> sop(O), factor(F2).
termP(F,T) --> sop(O), factor(F2), termP(expr(O,F,F2),T).

% Weak and Strong operators -- strong has higher
% precedence than weak

wop(+) --> [+].
wop(-) --> [-].
sop(*) --> [*].
sop(/) --> [/].
sop('%') --> ['%'].

% These are the basic units of expression.
% These are the basic units of expression.
factor(Num) --> num_constant(Num).
factor(neg(Num)) --> ['-'], num_constant(Num).
factor(Id) --> identifier(Id).
factor(neg(Id)) --> ['-'], identifier(Id).
factor(E) --> ['('], expression(E), [')'].
factor(fcall(F,Par)) -->
        [F], {atom(F)}, ['('], actuals(Par), [')'].
factor(fcall(F,[])) -->
        [F], {atom(F)}, ['('], [')'].
factor(if(B,E1,E2)) -->
        [if],['('], bool_expr(B), [')'], [':'],
```

```
        expression(E1), [else], [':'], expression(E2).

%% actual parameters
actuals([E]) --> expression(E).
actuals([E1|E2]) --> expression(E1), [','], actuals(E2).

num_constant(C) --> [C], {number(C)}.
identifier(Id) --> [Id], {atom(Id)}.

% (boolean_expression grammar) --
% A boolean expression is either an op expression,
% an and expression,
% or a negation expression
bool_expr(B) --> or_boolean(B).

% (or_boolean grammar) --
% A or_boolean is the boolean expression with the lowest precedence
% It is either an and_boolean expression
% or it is a boolean expression. followed by ||, followed by a second
% boolean expression
or_boolean(S) --> and_boolean(S).
or_boolean(TP) -->
        and_boolean(F),
        orPrime(F,TP).

% (orPrime grammar) --
% opPrime is used to remove left recursion
orPrime(In, bexp2(In,Op,F)) -->
        or_op(Op),
        and_boolean(F).
orPrime(In, Out) -->
        or_op(Op),
        and_boolean(F),
        orPrime(bexp2(In,Op,F),Out).

% (and_boolean grammar) --
% An and_boolean is the boolean expression with middle precedence
% It is either a not_boolean expression
% or it is a boolean expression. followed by &&, followed by a second
% boolean expression
and_boolean(S) --> not_boolean(S).
and_boolean(TP) -->
        not_boolean(F),
        andPrime(F,TP).

% (andPrime grammar) --
% andPrime is used to remove left recursion
andPrime(In, bexp2(In,Op,F)) -->
        and_op(Op),
        not_boolean(F).
andPrime(In, Out) -->
        and_op(Op),
        not_boolean(F),
        andPrime(bexp2(In,Op,F),Out).

% (not_boolean grammar) --
% A not_boolean expression is either a comparison or
```

```
% a negated comparison or
% a negated boolean expression in parenthesis
not_boolean(U) --> comparison(U).
not_boolean(bexp1(Op,B1)) -->
        not_op(Op),
        comparison(B1).
not_boolean(bexp1(Op,B1)) -->
        not_op(Op),
        ['('],
        or_boolean(B1),
        [')'].

not_op(lnot) --> ['!'].
and_op(land) --> ['&&'].
or_op(lor) --> ['||'].

% (definitions) --
% variable definition
% function definition
% procedure definition
definition(vdef(Name, Exp)) -->
  [var], identifier(Name), [':'], expression(Exp), [';'].
definition(fdef(Name, Params, Exp)) --> % with parameters
  [fun], identifier(Name), ['('], formals(Params), [')'],
  expression(Exp), [';'].
definition(fdef(Name, [], Exp)) --> % without parameters
  [fun], identifier(Name), ['('], [')'], [':'],
  expression(Exp), [';'].
definition(pdef(Name, Params, Blk)) --> % with parameters
  [proc], identifier(Name), ['('], formals(Params), [')'],
  block(Blk).
definition(pdef(Name, [], Blk)) --> % with parameters
  [proc], identifier(Name), ['('], [')'],
  block(Blk).

%% one or more definitions
definitions([Def|Rest]) --> definition(Def), definitions(Rest).
definitions([Def]) --> definition(Def).

%% formal parameters for function and procedures
%% note that var parameters only make sense for
%% procedures
formal(var(Id)) --> [var], identifier(Id).
formal(val(Id)) --> identifier(Id).

formals([F|Rest]) --> formal(F), [','], formals(Rest).
formals([F]) --> formal(F).

%% (statements) --
%% assignment | if | while | block | pcall | print | break
statement(S) --> assignment(S).
statement(S) --> if(S).
statement(S) --> while(S).
statement(S) --> block(S).
statement(S) --> break(S).
statement(S) --> pcall(S).
statement(S) --> print(S).
```

```
%% one or more statements
statements([S]) --> statement(S).
statements([S|Rest]) --> statement(S), statements(Rest).

assignment(assign(Var,RHS)) -->
  identifier(Var), ['='], expression(RHS), [';'].
if(if1(B,S)) -->
  [if], ['('], bool_expr(B), [')'], statement(S).
if(if2(B,S1,S2)) -->
  [if], ['('], bool_expr(B), [')'], statement(S1),
  [else], statement(S2).
while(while(B,S)) -->
  [while], ['('], bool_expr(B), [')'], statement(S).
block(block(D,S)) -->
  ['{'], local_vars(D), statements(S), ['}'].
block(block([],S)) -->
  ['{'], statements(S), ['}'].
pcall(pcall(P, Par)) -->
  [P], {atom(P)}, ['('], actuals(Par), [')'], [';'].
pcall(pcall(P, [])) -->
  [P], {atom(P)}, ['('], [')'], [';'].
print(print(E)) --> [print], expression(E), [';'].
break(break) --> [break], [';'].

%% a block may contain local variables
local_var(vdef(Name, Exp)) -->
  [var], identifier(Name), [':='], expression(Exp), [';'].

local_vars([Def|Rest]) --> local_var(Def), local_vars(Rest).
local_vars([Def]) --> local_var(Def).

% A program in C-- is a list of zero or more definitions followed
% by a call to main().
program(prog(Defs, pcall(main,[]))) --> definitions(Defs), [main], ['('],
[')'], [';'].
```