Key:

* [ ] means optional
* \* means infinitely repeatable
* E is the empty set
* Red text is terminal
* Purple text is semantic action

<program> -> <functions> {return functions.n}

<functions> -> <declarations> <function> [<declarations><function>]\* {functions.n = new Seq(function.n, functions.n)}

<function> -> breakfastOrder id([<declaration>], [<declaration>]\* ) [orderOf <type>]

<block> {function.n = new function(params.list, block.n, ret.type)}

<block> -> { [<declarations>] [<stmnts>]} {return stmnts.n}

<stmts> -> <stmnt> [<stmnt>]\* {return new Seq(stmnt.n, stmnts.n)}

<stmnt> -> orderUp (<expr> | E); {return new OrderUp(expr.n)}

| id <- <expr>;

{

if(isNumeric(id.type) && !isNumeric(expr.type)) throw Error;

else if(isBool(id.type) && !isBool(expr.type)) throw Error;

return new Assign(st.getIdFromToken(id.token), expr.n)}

}

| break; {return new Break()}

| roll(id : (<expr>) -> (<expr>) stepBy (<expr>))<block>

{

if(!Type::isNumericType(<expr>)) throw Error;

return new Roll(id.token, expr.n, expr.n, expr.n, block.n);

}

| fresh(<bool>) <block> [?fresh(<bool>) ]\* [expired <block>]

{

if(bool.type != bool) throw Error;

return new Fresh(bool.n, block.n, else.n)

}

<expr> -> <mult> {return mult.n}

| id {return st.getIdFromToken(id)}

| <factor> {return factor.n}

| <sum> {return sum.n}

<mult> -> <expr> (\* | / ) (<expr> | (<sum>))

{

if(!isNumeric (expr) || !isNumeric(<expr>)) throw Error;

expr.type = greaterType(expr.type, expr.type); //Coerce the types

return new Mult(‘+/-’, expr.n, expr.n)

}

<sum> -> <expr> (+ | -) ((<expr>))

{

if(!isNumeric (expr) || !isNumeric(<expr>)) throw Error;

expr.type = greaterType(expr.type, expr.type); //Coerce the types

return new Sum(‘+/-’, expr.n, expr.n)

}

<declarations> -> <declaration> [<declaration>]\*

<declaration> -> <type> @ id; {st.add(id.token, new Id(id.token, type.n)}

<bool> -> (bacon | sausage | <equality> | <logicBool>) [ || <bool>]

<logicBool> -> <bool> && <bool>

{

if(bool.type != type::bool) throw Error;

return new logicBool(‘&&’, bool.n, bool.n)

}

<equality> -> <bool> <equalityOp> <bool> {

if(bool.type != type::bool) throw Error;

return new equality(equalityop.token, bool.n, bool.n)

}

<equalityOp> -> <==> | <=!=>

<factor> -> <bool> {return bool.n} | num {return new Const(num.value, num.type)}

| real {return new Const(real.value, real.type)} | char {return new Const(char.value, char.type)}

| str {return new Const(str.value, new Array(str.type, str.length)}

<type> -> breakfastItem | wholeMilk | percentMilk | baconOrSausage | <array>

<array>-> <type>[<expr>]

{

if(!isNumeric(expr.type)) throw Error;

return new Array(type.n, expr.n)

}