Listing 1: Solve

```csharp

using Expi;
using Parsec;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Constro.Parser
{
    public class SolvePAA: ContextStatement
    {
        ContextArguments arguments;

        public SolvePAA(ContextArguments arguments)
        {
            this.arguments = arguments;
        }

        public void Execute(Context context)
        {
            var args = arguments.Value(context);
            var nec = (string)Arg("nec", "n", args);
            var theta = (double)Arg("theta", null, args);
            var phi = (double)Arg("phi", null, args);
            var package = (string)Arg("package", "p", args);
            var solver = (string)Arg("solver", "s", args);
            var compare = (string)Arg("compare", "c", args);
            DateTime start = DateTime.Now;
            Console.WriteLine($"Started at {start}");
            bool draw = true;
            if (args.ContainsKey("draw"))
            {
                draw = bool.Parse((string)args["draw"]);
                args.Remove("draw");
            }
            var exp = new Experiment(nec, package, solver, theta, phi,
                compare, draw, args);
            exp.Solve(false);
            DateTime end = DateTime.Now;
            Console.WriteLine($"Finished at {end}");
            double mils = (end - start).TotalMilliseconds;
            Console.WriteLine($"Duration: {mils / 1000} s.");
        }

        object Arg(string l, string s, Dictionary<string, object> args)
        {
            if (args.ContainsKey(l))
            {
                var a = args[l];
                args.Remove(1);
                return a;
            }
            if (s != null && args.ContainsKey(s))
            {
                var a = args[s];
                args.Remove(s);
                return a;
            }
            throw new NoParameterException();
        }
    }

    public class SolveParser : ConstroWordsParser
    {
        public SolveParser() : base(new Map<ContextArguments,
            ContextStatement, StringBuilder>(
            ArgsAfter(Words("solve", "paa")),
            a => new SolvePAA(a)
        )) {}
    }
}
```

Listing 2: Wire

```
using Parsec;
using System.Collections.Generic;
using System.Numerics;
using System.Text;

namespace Constro.Parser
{

    public class WireStatement : ContextStatement
    {
        public static string WiresKey = "${wires}";
        ContextWire Wire;

        public WireStatement(ContextWire Wire)
        {
            this.Wire = Wire;
        }

        public void Execute(Context context)
        {
            List<SegmentedWire> segmentedWires = (List<SegmentedWire>)
                context.LocalValue(WiresKey);
            if (segmentedWires == null)
            {
                segmentedWires = new List<SegmentedWire>();
            }
            var wire = Wire.Value(context);
            segmentedWires.Add(wire);
            context.Set(segmentedWires, WiresKey);
        }
    }

    public class WireStatementParser: CustomParser<ContextStatement,
        StringBuilder>
    {
        public WireStatementParser() : base(
            new Map<ContextWire, ContextStatement, StringBuilder>(
                new WireParser(),
                wire => new WireStatement(wire)
            )
        ) { }
    }
    public class ContextJunction: ContextExpression<Junction>
    {
        public ContextJunction(string str = "->") : base(str) { }

        public override Junction Value(Context context)
        {
            return new SimpleJunction();
        }
    }

    public class ContextFedJunction : ContextJunction
    {
        ContextExpression<Complex> value;
        FedJunction.Mesure mesure;
        public ContextFedJunction(ContextExpression<Complex> value,
            FedJunction.Mesure mesure) : base($"~{value} {mesure}~")
        {
            this.value = value;
            this.mesure = mesure;
        }

        public override Junction Value(Context context)
        {
            var complex = this.value.Value(context);

            return new FedJunction(complex, mesure);
        }
    }

    public class JunctionParser: CustomParser<ContextJunction,
        StringBuilder>
    {
        public JunctionParser() : base(
            new Map<StringBuilder, ContextJunction, StringBuilder>(
                new StringParser("->").SkipLeadingWhitespaces,
                (str) => {
                    return new ContextJunction();
                }
            )
        ) { }
    }

    public class FedJunctionParser : CustomParser<ContextFedJunction, StringBuilder>
    {
        static GenericParser<StringBuilder, StringBuilder> TildaParser = new StringParser("~").SkipLeadingWhitespaces;
        static GenericParser<FedJunction.Mesure, StringBuilder> MesureParser = new Or<FedJunction.Mesure, StringBuilder>(
            new Map<StringBuilder, FedJunction.Mesure, StringBuilder>(
                new SkipLeadingWhitespaces(new CharSetParser("vV").StringParser),
                (str) => { return FedJunction.Mesure.Voltage; }
            ),
            new Map<StringBuilder, FedJunction.Mesure, StringBuilder>(
                new SkipLeadingWhitespaces(new CharSetParser("aA").StringParser),
                (str) => { return FedJunction.Mesure.Current; }
            )
        );

        static GenericParser<Pair<ContextComplex, FedJunction.Mesure>, StringBuilder> ValueParser = new Both<ContextComplex, FedJunction.Mesure, StringBuilder>(
            new ContextComplexParser(),
            MesureParser
        );

        static GenericParser<ContextFedJunction, StringBuilder> JunctionParser =
            new Map<Pair<ContextComplex, FedJunction.Mesure>, ContextFedJunction, StringBuilder>(
                ValueParser,
                (value) =>
                {
                    return new ContextFedJunction(value.First, value.Second);
                }
            );

        public FedJunctionParser() : base(
            new Right<StringBuilder, ContextFedJunction, StringBuilder>(
                TildaParser,
                new Left<ContextFedJunction, StringBuilder, StringBuilder>(
                    JunctionParser,
                    TildaParser
                )
            )
        )
        { }
    }

    public class ContextWire: ContextExpression<SegmentedWire>
    {
        ContextPoint anchor;
        Pair<ContextJunction, ContextPoint>[] segments;

        public ContextWire(ContextPoint anchor, Pair<ContextJunction,
            ContextPoint>[] segments) : base($"{anchor} -> ...")
        {
            this.anchor = anchor;
            this.segments = segments;
        }

        public override SegmentedWire Value(Context context)
        {
            var segments = new List<Segment>();
            var segs = this.segments;
            if (segs.Length == 0)
            {
                return new SegmentedWire(segments.ToArray());
            }
            var t = MakeTransform(context);
            var segment = MakeSegment(anchor, segs[0].First, segs[0].
                Second, context, t);
            segments.Add(segment);
            for(var i = 0; i < segs.Length - 1; i++)
            {
                var lhs = segs[i];
                var rhs = segs[i + 1];
                segment = MakeSegment(lhs.Second, rhs.First, rhs.Second,
                    context, t);
                segments.Add(segment);
            }
            return new SegmentedWire(segments.ToArray());
        }

        private Transform MakeTransform(Context context)
        {
            var initial = Transform.Identity();
            if (context.Parent != null)
            {
                initial = MakeTransform(context.Parent);
            }
            var localObj = context.LocalValue(TransformStatement.
                TransformKey);
            if (localObj == null)
            {
                return initial;
            }
            var local = (Transform)localObj;
            var t = initial * local;
            return t;
        }

        private Segment MakeSegment(ContextPoint left, ContextJunction
            junction, ContextPoint right, Context context, Transform t)
        {
            var lhs = t * left.Value(context);
            var rhs = t * right.Value(context);
            var junc = junction.Value(context);
            return new Segment(lhs, junc, rhs);
        }
    }

    public class WireParser: CustomParser<ContextWire, StringBuilder>
    {

        static GenericParser<ContextJunction, StringBuilder> JuctParser =
            new Or<ContextJunction, StringBuilder>(
            new JunctionParser(),
            new Map<ContextFedJunction, ContextJunction, StringBuilder>(
                new FedJunctionParser(), j => j
            )
        );
        static GenericParser<Pair<ContextJunction, ContextPoint>,
            StringBuilder> SegmentParser = new Both<ContextJunction,
            ContextPoint, StringBuilder>(
            JuctParser, new PointParser()
        );

        static GenericParser<Pair<ContextJunction, ContextPoint>[],
            StringBuilder> SegmentsParser = new ManyOne<Pair<
            ContextJunction, ContextPoint>, StringBuilder>(
            SegmentParser
        );

        public WireParser() : base(
            new Map<Pair<ContextPoint, Pair<ContextJunction, ContextPoint
                >[]>, ContextWire, StringBuilder>(
                new Both<ContextPoint, Pair<ContextJunction, ContextPoint
                    >[], StringBuilder>(
                    new PointParser(),
                    SegmentsParser
                ), pair => new ContextWire(pair.First, pair.Second)
            )
        ) { }
    }
}
```

Listing 3: Def

```
using Parsec;
using System.Collections.Generic;
using System.Text;

namespace Constro.Parser
{
    public class Def<A> : ContextStatement
    {
        protected string name;
        ContextExpression<A> value;

        public Def(string name, ContextExpression<A> value)
        {
            this.name = name;
            this.value = value;
        }

        public virtual void Execute(Context context)
        {
            context.Set(this, name);
        }

        public A Unwrap(Context context)
        {
            return value.Value(context);
        }
    }

    public class UnwrappingDef<A> : Def<A>
    {

        public UnwrappingDef(string name, ContextExpression<A> value):
            base(name, value)
        {
        }

        public override void Execute(Context context)
        {
            context.Set(Unwrap(context), name);
        }
    }

    public class ContextGroup: ContextExpression<Group>
    {
        public string Name;
        public ContextStatement[] Content;

        public ContextGroup(string name, ContextStatement[] content) :
            base(name)
        {
            this.Name = name;
            this.Content = content;
        }

        public override Group Value(Context context)
        {
            foreach (var statement in Content)
            {
                statement.Execute(context);
            }
            var list = (List<SegmentedWire>)context.Value(WireStatement.
                WiresKey);
            if (list == null)
            {
                return new Group(Name, new SegmentedWire[] { });
            }
            return new Group(Name, list.ToArray());
        }
    }

    public class DefParser<A>: ConstroWordsParser
    {
        protected static GenericParser<string, StringBuilder> IdentifierParser =
            new Map<StringBuilder, string, StringBuilder>(
                IdAfter("def"),
                str =>
                {
                    return str.ToString();
                }
            );

        protected static GenericParser<string, StringBuilder> AssignParser = new Left<string, StringBuilder, StringBuilder>(
            IdentifierParser,
            Word("=")
        );

        public DefParser(GenericParser<ContextStatement, StringBuilder> parser): base(parser)
        {

        }
    }

    public class DefDoubleParser: DefParser<double>
    {
        static GenericParser<ContextStatement, StringBuilder> DoubleParser =
            new Map<Pair<string, ContextExpression<double>>, ContextStatement, StringBuilder>(
                new Both<string, ContextExpression<double>, StringBuilder>(
                    AssignParser,
                    new ContextDoubleExpressionParser()
                ),
                pair => new UnwrappingDef<double>(pair.First, pair.Second)
            );
        public DefDoubleParser() : base(DoubleParser) {
        }
    }

    public class DefPointParser: DefParser<Position>
    {
        static GenericParser<ContextStatement, StringBuilder> PointParser =
            new Map<Pair<string, ContextPoint>, ContextStatement, StringBuilder>(
                new Both<string, ContextPoint, StringBuilder>(
                    AssignParser,
                    new PointParser()
                ),
                pair => new UnwrappingDef<Position>(pair.First, pair.Second)
            );

        public DefPointParser() : base(PointParser)
        {
        }
    }

    public class DefGroupParser: DefParser<Group>
    {
        public DefGroupParser(IdentifiersParser identifiersParser) : base(
            Statements(identifiersParser, IdentifierParser, (i, s) =>
            {
                identifiersParser.Register(i);
                return new Def<Group>(i, new ContextGroup(i, s));
            })
        )
        { }

        public override ResultOrError<ContextStatement, StringBuilder> Parse(StringBuilder input)
        {
            return base.Parse(input);
        }
    }
}
```