Silver and Copper Tutorial

MELT Group
University of Minnesota
`http://melt.cs.umn.edu`

Draft of December 30, 2010 at 20:57

Contents

1 Introduction

This tutorial provides a brief introduction to Silver, an extensible attribute grammar system, and Copper, a parser and context-aware scanner generator. These tools have been developed by the MELT group at the University of Minnesota for specifying extensible languages and generating compilers and translators from these language specifications.

This tutorial is intended for readers with no experience in building language processors such as compilers, source-to-source translators, or interpreters. It assumes that readers are familiar with basic notions from programming languages such as types, control flow, variable declaration, etc. Readers need have no experience with building language processors or with for generating language processors. This includes students using Silver and Copper for the first time.

This tutorial may be too remedial for readers with have experience using grammarware tools. Such tools include parser and scanner generators, such as Yacc/Lex, Bison/Flex, ANTLR, ASF+SDF, SGLR, or Copper, attribute grammar systems, such as The Synthesizer Generator, JastAdd, LRC, or Silver, and term rewriting system, such as ASF+SDF and Stratego.

This tutorial is also not self contained. It has numerous pointers to other sources of information as it makes no sense to duplicate perfectly good descriptions of relevant topics found in other sources.

The primary goal of this document is to help new users begin using Silver and Copper. In Section 2 we describe the components in a traditional compiler pipeline: scanner, parser, semantic analyzer, and code generator. In the following sections we describe how to specify each of this using Silver and Copper.

2 The architecture and components of a language translator.

A compiler or source-to-source translator are tools that read in a text file containing a program and write out another text file containing the translation of that program. For compilers, that generated program is in a low-level language such as assembly language, byte-codes for a virtual machine such as the Java virtual machine (JVM), or sometimes even C. For source-to-source translators the generated program is a higher-level language, which may or may not be the same language as the input program. Thus, the names “compiler” is a bit of a misnomer; it is simply a translator that generates low-level code. Compilers and source-to-source translators are essentially the same kinds of tools built using the same techniques.

Throughout this document we will use the term “translator” to indicate any tool that translates a program from one language, the source language, to another, the target language. In some cases the source and target languages are the same; for example a tool that reformats a program to comply with strict formatting guidelines.

It is common practice to construct language translators as a series or pipeline of components that perform the major tasks in program translation.

Scanning: The scanning process reads in program text and recognizes its lexical syntax. This process involve recognizing lexical constructs such as keywords (*e.g.* `while` or `if`), identifier names (*e.g.* `x`, `area`, etc.), literal values (*e.g.* `1`, `3.14`, `"Hello"`, etc.) and generating a sequence of *tokens*, one for each such recognized construct. This sequence of tokens is passed on to the parsing phase that follows. The scanner is also responsible for recognizing and removing comments from the input stream, for these no token is generated.

For example, from the input stream "`while ( x < 100 ) x = x * x ;`" the following list of tokens may be generated:

[While_t("`while`"), LParen_t("`(`"), Id_t("`x`") , LT_t("`<`") , IntLit_t("`100`"), RParen_t("`)`"), Id_t("`x`") , EQ_t("`=`"), Id_t("`x`") , Star_t("`*`"), Id_t("`x`") , Semi_t("`;`")]

Parsing: The parsing process recognizes the syntactic structure in the sequence of tokens produced by the scanner. This structure is represented as a tree. For the example the parser would create a tree with a root node labeled to indicate that it was a while-loop and with two children; the first being the tree representing the while-loop condition and the second being the tree representing the while-loop body.

This tree is called the *concrete syntax tree* since it is based on the specification of the actual concrete syntax, as opposed to a simplified abstract syntax that is sometimes used in the following semantic analysis phase.

Semantic analysis: This phases examines a tree representation of the program to perform analysis such as type checking and error reporting.

Code generation: This is the final phase in which some translation of the tree representation is generated. This may be byte-code, machine code, or a translation to a language such as C. In most compilers this phase is preceded by an optimization phase.

3 Scanning

The role of the scanner is to take the program text as input and recognize the keywords, constants, operator symbols and other items in the input text. It also recognizes comments and discards them as they are not needed for processing the program.

The output of the scanner can be thought of as sequence of tokens; each indicating the type of symbol recognized, the string in the input that was recognized, and, typically, a line and column number.

The "type" of symbols that are recognized in a language is determined by specifying a set of *terminal symbols*, simply called *terminals.* For example, a terminal may be defined that matches integer constants, another for program variable names, another for the "while" keyword in an imperative language. To specify this, terminals are defined by a name and a regular expression defines the strings the terminal symbol should match. We may define a terminal symbol named Int_t with a regular expression that matches a sequence of 1 or more digits.

A token generated for a sequence of characters is an object that specifies what type of token, the text that was recognized as generating the token (called the *lexeme*), and in many cases, the line and column number of the file where the lexeme is located.

A scanner is constructed from the specifications of terminal symbols in the grammar. Terminals symbols are given names which must be capitalized since terminals are essentially types in Silver and type name must be capitalized.

Regular expressions associated with a terminal are written after the terminal name between forward slashes. The file `Terminals.sv` in the tutorial grammar `dc` contains the following declaration for integer literal terminals.

```
terminal IntLit_t /[0-9]+/ ;
```

Regular expressions that are constant strings may be represented using single quotes, as seen in the declaration of the plus symbol in the same file:

```
    terminal Plus_t '+' ;
```

Terminal declarations have the form

terminal ⟨*name*⟩ (/⟨*regex*⟩/ | '⟨*string*⟩') ⟨*optional-claues*⟩ ;

Terminals in which the regular expression is written between single quotes require that the regular expression be a simple string that when considered as a regular expression will match only that string. In Silver specifications we can then refer to that terminal symbol using the quoted string instead of its name.

Consider a simple imperative language, such as Simple defined in the tutorial grammar `simple`. Because the regular expression for the keywords such as $While_t$, If_t, and $Else_t$ all overlap with the language defined by the regular expression for identifiers Id_t we need some mechanism to disambiguate them. This is done by giving the keyword terminal *lexical precedence* over the identifier terminal. Thus the string "`while`" which matches both the regular expression of $While_t$ and Id_t will be matched with $While_t$ as we would expect. Lexical precedence can be specified in two way. First, as done in the example, an optional clause of the form

submits to { ⟨*comma-separated-names*⟩ }

indicates that the terminal being defined has lower lexical precedence than those listed. A second form is

dominates { ⟨*comma-separated-names*⟩ }

indicates the opposite, that the terminal being defined has higher lexical precedence than those in the list. The names between curly braces may be the names of terminal symbols or of *lexer classes*, to which terminals may claim membership and thus be considered in any lexical precedence relations specified using the lexer class name. Examples of this can be found in the `Terminals.sv` file in the tutorial grammar `simple:terminals`, a few of which are copied below:

```
lexer class keywords ;

terminal If_t    'if'     lexer classes { keywords } ;
terminal Else_t  'else'   lexer classes { keywords } ;
terminal While_t 'while'  lexer classes { keywords } ;

terminal Id_t  /[a-zA-Z][a-zA-Z0-9_]*/  submits to { keywords } ;
```

Here, keywords specify that they are members of the `keywords` class. The identifier terminal indicate that is has lower lexical precedence than keywords as desired.

Terminals declarations with the optional leading `ignore` keyword are recognized by the scanner but then dropped and not returned to the parser. Below are the specifications of white space and line comments from `simple`, both of which are recognized by the scanner but not returned to the parser.

```
ignore terminal WhiteSpace_t /[\t\n\ ]+/  ;      -- white space
ignore terminal LineComment_P  /[\/][\/].*/  ;  -- line comments
```

4 Parsing

Parsing is the process of discovering the syntactic structure of the program in the linear (unstructured) sequence of tokens generated by the scanner. This syntactic structure is a tree in which tokens form the leaves of the tree. Interior nodes of the tree have types, called *nonterminals*, the specify certain grammatical categories. In an imperative programming language we may have nonterminals for statements, expressions, declarations, etc. These nodes are created by tree-constructing functions that take trees as input that become the child trees of the created tree.

Context free grammars are the formalism that is almost universally used for specifying the syntactically valid tree structures of a language. Grammars have the form

$$G = \langle NT, T, P, S \rangle$$

where

- NT is a finite set of nonterminal symbols
- T is a finite set of terminal symbols
- P is a finite set of productions specifying how new trees can be created. These have the form $NT \to (NT \mid T)*$; a nonterminal on the left hand side specifying type of the created tree and the possibly empty sequence of nonterminals and terminals on the right hand side specifying the types of the child trees.
- S is the grammars start symbol that indicates the type of nonterminal at the root any tree representing a complete program. $S \in NT$.

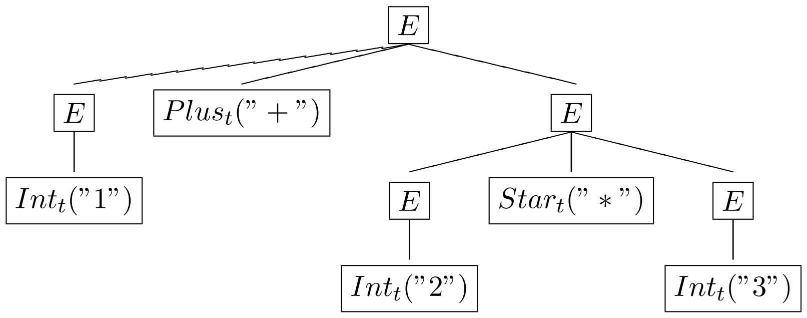
Consider the following grammar for arithmetic expressions:

$$\begin{aligned}
G = \langle\ & NT = \{E\},\ \ T = \{Int_t,\ Plus_t, Star_t, LP_t, RP_t\text{`}+\text{'},\ \text{`}*\text{'},\ \text{`('},\ \text{`)'}\},\\
& P = \{\ E \to E\ \text{`}+\text{'}\ E,\\
& \qquad\ \ E \to E\ \text{`}*\text{'}\ E,\\
& \qquad\ \ E \to \text{`('}\ E\ \text{`)'},\\
& \qquad\ \ E \to Int_t\}\\
& S = E\rangle
\end{aligned}$$

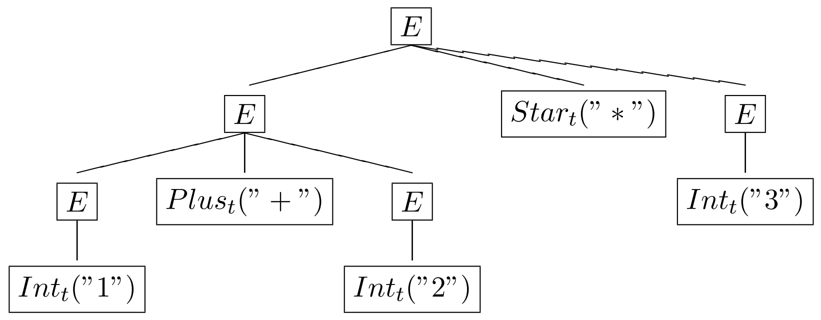
and the following sequence of tokens, possibly produced by a scanner.

$$[\ Int_t(\text{"1"}),\ Plus_t(\text{"}+\text{"}),\ Int_t(\text{"2"}),\ Star_t(\text{"}*\text{"}),\ Int_t(\text{"3"})\ ]$$

The following syntax tree is a valid "parse tree" for this sequence of tokens as the tree conforms to the restrictions placed on the grammar.



While this tree is a perfectly fine representation of the structure in the sample expression, the following tree also conforms to the grammar and can be generated from the same set of sample tokens:



The problem is that the grammar given above is *ambiguous*. Such grammars have more than one tree for a given input and are thus not suited as grammars from which parsers can be built. The tree directly above has a shape that suggests that the addition will be done before the multiplication, and this violates the traditional operator precedence rules of arithmetic.

What is needed is a grammar that is non-ambiguous and enforces the rules of operator precedence and associativity. The grammar in Figure 1 satisfies these constraints and is written in the syntax of Silver. Silver syntax is somewhat different from the syntax of other parser generators. This is because it can be used in more general manner since all of the attribute grammar specifications in Silver can be applied to concrete productions.

- Productions are named. These names are not often used for concrete productions, those used by Copper to generate the parser, but they may be. The productions here all have the `concrete` modifier and are thus passed to Copper to be used in generating the parser. Abstract productions, indicated by the `abstract` keyword are used for semantic analysis as described in Section 5 and not passed to Copper for use in parser generation.

  Semantic actions are allowed on concrete productions and are written between the curly braces. In Figure 1 these are elided.

- When terminals are constant their regular expression can be written using single quotes, as described above. In this case they can be referenced by the quoted string instead of the name of the terminal. This sometimes makes for easier reading of the productions.

Parser specification and use: A parser is generated from a grammar (or collection of them) and the name of the nonterminal that is the *start symbol* for the grammar. This is the nonterminal that will appear at the root to the concrete syntax tree returned by the parser.

In `Main.sv` the following specification specifies a parser named `parse`

```
parser parse :: Root_c { dc; }.
```

The generated parser `parse` is a function that takes two strings, the text to parse and name of the file which is then used in any error messages generated by the parser. This function returns a result of type `ParseResult<Root_c>`. This is a parameterized nonterminal that defines three attributes:

1. `parseSuccess` which is `true` if the text was successfully parsed

2. `parseTree` which is the tree returned if the parse was successful. For `parse` this is a tree of type `Root_c`.

```
nonterminal Root_c ;

concrete production root_c
r::Root_c ::= e::Expr_c   { }

nonterminal Expr_c, Term_c, Factor_c ;

concrete production add_c
sum::Expr_c ::= e::Expr_c '+' t::Term_c  { }

concrete production sub_c
dff::Expr_c ::= e::Expr_c '-' t::Term_c  { }

concrete production exprTerm_c
e::Expr_c ::= t::Term_c   { }

concrete production mul_c
prd::Term_c ::= t::Term_c '*' f::Factor_c  { }

concrete production div_c
d::Term_c ::= t::Term_c '/' f::Factor_c  {  }

concrete production termFactor_c
t::Term_c ::= f::Factor_c  { }

concrete production nested_c
e::Factor_c ::= '(' inner::Expr_c ')'   { }

concrete production integerConstant_c
ic::Factor_c ::= i::IntLit_t   { }
```

Figure 1: Concrete syntax of `dc`.

3. `parseErrors` which is a string containing a description of the parse errors if the parse was not successful.

The file `Main.sv` also makes use of Silver's rather esoteric input output constructs defined in Section 6.

Operator precedence and associativity specifications: Copper, like many parser generators, also supports the specification of operator precedence and associativity as annotations on terminal declarations. Examples of these types of specifications can be found in the `Terminals.sv` file of the tutorial grammar `simple:terminals` and are explained in more detail in the Silver and Copper reference manuals.

LALR(1) parsing and parser conflicts: Copper generates LALR(1) parsers and context-aware scanners. Many compiler textbooks describe LALR(1) parsing and the challenges of specifying a grammar so that it falls into the LALR(1) class of grammars. When Copper reports a conflict, it is because the grammar is not LALR(1) and needs to be refactored to be in this class.

Readers that are serious about designing scanner and parsers need to read about LR parsing (which include LR(1) and LALR(1) parsing) to understand the underlying parsing algorithms.

5 Attribute Grammars

Attribute grammars are formal specifications that can be used to define the semantics, a *meaning* of some sort, to a program, expression, or phrase, in a language. Attribute grammars define semantics by specifying that certain semantic attributes (values) will be associated with certain nodes in the syntax tree and providing equations that specify how the values of these attributes are to be computed. Don Knuth's 1968 paper "Semantics of Context Free Languages" [3] introduced attribute grammars and is essential reading.

Attributes may be associated with nonterminals in the grammar. Attributes that are used to propagate semantic information up the syntax tree are called *synthesized* attributes and attributes that propagate information down the tree are called *inherited* attributes. In a AG defining the semantics of a programming language, an example of a synthesized attribute may be an *errors* attribute that collects semantic errors and propagates them to the root of the tree so that they can be reported to the programmer. An inherited attribute may be a symbol-table or environment that associates variable names with their types. This information is propagated down the trees representing statements and expression so that it can be used to determine the types of variable uses and then be used in type checking.

5.1 Synthesized attributes

In Silver, attribute declarations have the following form:

(`inherited` | `synthesized`) `attribute` ⟨*attr-name*⟩ `::` ⟨*attr-type*⟩ ⟨*optional-attr-clauses*⟩ `;`

For example, consider the declaration of a string attribute that "unparses" the tree, generating a string representation of it named `pp`.

```
synthesized attribute pp :: Sting ;
```

Since this attribute is synthesized, its value on a node is computed, typically, from values of this attribute one the nodes children. Values of attributes on nodes are written using the name of the node as indicated in the signature of the production and the name of the attribute separated by a dot ("`.`"). Figure 2 fills in the definitions of the `pp` attribute that were left out of Figure 1. The attribute `pp` is declared as a `String` attribute and it decorates all four nonterminals. On the root production, the value of `pp` on the root node `r` is the same as the value of the `pp` attribute on its single child `e`. On the production for addition the string concatenation operator `++` is used to compute the value of `pp` on the node named `sum` from the value of `pp` on its children and string constants. Sting constants in Silver are represented as they are in most other programming languages. On the production `integerConstant_c` we see that the value of `pp` is copied from the lexeme of the integer literal terminal symbol named `i` in this production. Terminal are also decorated with `Integer`-valued attributes `line` and `column` indicating where in the input file they were matched.

Silver supports attributes of various types. Primitive types supported by Silver include `Integer`, `Float`, `String`, and `Boolean`. The expected arithmetic (`+`, `*`, etc) and relational (`<`, `!=`, etc.) operators are support on numeric values. Logical (`&&`, `||`, `!`) operators are supported on boolean values. The string concatenation operator `++` and numerous library functions on strings are also supported. The reader is referred to the Silver Reference Manual for a full accounting of these.

Silver also supports attributes whose values are syntax trees. These are called *higher-order* attributes [7] but are not as complicated as this name might indicate. Productions are essentially tree building functions that take trees or other values that become the child nodes of the root node in the constructed tree. The parser generated by Copper in essence uses concrete productions as tree building functions to construct the concrete syntax tree.

One important use of higher-order attributes is in creating an abstract syntax tree from the concrete syntax tree. An abstract syntax tree is typically based on a simpler grammar that can omit punctuation and operator symbols that are only needed in recognizing the structure of the program from the linear sequence of tokens produced by the scanner. This grammar is also usually simpler because it does not need to conform to any parser-imposed restrictions. Abstract productions are not used by Copper in generating the parser. The abstract syntax of `dc` can be seen in Figure 3. Abstract productions are indicated by the modifier `abstract`. These productions define values of `pp` and an integer valued `value` attribute that computes the value of the expression.

Figure 4 shows some the declaration of the higher-order attributes `ast_Expr` and `ast_Root` some of the concrete productions with their definitions of these attributes. For example, on `add_c` we see that the abstract syntax tree from addition is built using the abstract production `add` and the abstract syntax trees of the two expression children; the operator symbol is not present in the abstract syntax since it is no longer need. One the production `exprTerm_c` the abstract syntax tree of the child `t` is used as the abstract syntax tree for `e` without modification; there is no corresponding `exprTerm` production in the abstract syntax since this production is used in the concrete syntax only to encode the operator precedence and associativity rules. Similarly, there are not abstract versions of the concrete nonterminals `Term_c` and `Factor_c`. There is only one abstract expression nonterminal, `Expr`, and an abstract root nonterminal `Root`.

Naming conventions: Note that concrete productions and nonterminals often have the suffix "`_c`" to distinguish them from their counterparts in the abstract syntax. This is only a naming convention and not part of the Silver specification language.

```
nonterminal Root_c ;
synthesized attribute pp :: String ;
attribute pp occurs on Root_c ;

concrete production root_c
r::Root_c ::= e::Expr_c
{ r.pp = e.pp;
}

nonterminal Expr_c with pp ;
nonterminal Term_c with pp ;
nonterminal Factor_c with pp ;

concrete production add_c
sum::Expr_c ::= e::Expr_c '+' t::Term_c
{ sum.pp = e.pp ++ " + " ++ t.pp ;  }

concrete production sub_c
dff::Expr_c ::= e::Expr_c '-' t::Term_c
{ dff.pp = e.pp ++ " - " ++ t.pp ;  }

concrete production exprTerm_c
e::Expr_c ::= t::Term_c
{ e.pp = t.pp ;  }

concrete production mul_c
prd::Term_c ::= t::Term_c '*' f::Factor_c
{ prd.pp = t.pp ++ " * " ++ f.pp ;  }

concrete production div_c
d::Term_c ::= t::Term_c '/' f::Factor_c
{ d.pp = t.pp ++ " / " ++ f.pp ;  }

concrete production termFactor_c
t::Term_c ::= f::Factor_c
{ t.pp = f.pp ;  }

concrete production nested_c
e::Factor_c ::= '(' inner::Expr_c ')'
{ e.pp = "(" ++ inner.pp ++ ")" ;  }

concrete production integerConstant_c
ic::Factor_c ::= i::IntLit_t
{ ic.pp = i.lexeme ;  }
```

Figure 2: Definition of the pp attribute on the concrete syntax productions.

```
nonterminal Root with pp;

synthesized attribute value :: Integer occurs on Root;

abstract production root
r::Root ::= e::Expr
{ r.pp = e.pp ;
  r.value = e.value ;
}

nonterminal Expr with pp, value;

abstract production add
sum::Expr ::= l::Expr r::Expr
{ sum.pp = "(" ++ l.pp ++ " + " ++ r.pp ++ ")";
  sum.value = l.value + r.value ;
}

abstract production mul
prd::Expr ::= l::Expr r::Expr
{ prd.pp = "(" ++ l.pp ++ " * " ++ r.pp ++ ")";
  prd.value = l.value * r.value ;
}

abstract production integerConstant
e::Expr ::= i::IntLit_t
{ e.pp = i.lexeme ;
  e.value = toInt(i.lexeme);
}
```

Figure 3: Some of the abstract grammar for `tutorial:dc` in file `AbstractSyntax.sv`.

```
nonterminal Root_c ;
synthesized attribute pp :: String ;
synthesized attribute ast_Root :: Root;
attribute pp, ast_Root occurs on Root_c ;

concrete production root_c
r::Root_c ::= e::Expr_c
{ r.pp = e.pp;
  r.ast_Root = root(e.ast_Expr);
}

synthesized attribute ast_Expr :: Expr ;
nonterminal Expr_c with pp, ast_Expr;
nonterminal Term_c with pp, ast_Expr;
nonterminal Factor_c with pp, ast_Expr;

concrete production add_c
sum::Expr_c ::= e::Expr_c '+' t::Term_c
{ sum.pp = e.pp ++ " + " ++ t.pp ;
  sum.ast_Expr = add(e.ast_Expr, t.ast_Expr );
}

concrete production exprTerm_c
e::Expr_c ::= t::Term_c
{ e.pp = t.pp ;
  e.ast_Expr = t.ast_Expr ;
}

concrete production mul_c
prd::Term_c ::= t::Term_c '*' f::Factor_c
{ prd.pp = t.pp ++ " * " ++ f.pp ;
  prd.ast_Expr = mul(t.ast_Expr, f.ast_Expr);
}

concrete production termFactor_c
t::Term_c ::= f::Factor_c
{ t.pp = f.pp ;
  t.ast_Expr = f.ast_Expr ;
}

concrete production integerConstant_c
ic::Factor_c ::= i::IntLit_t
{ ic.pp = i.lexeme ;
  ic.ast_Expr = integerConstant(i);
}
```

Figure 4: Concrete syntax specification of `dc` with higher order attribute generating the abstract syntax tree in file `ConcreteSyntax.sv`.

5.2 Inherited attributes

Inherited attributes are those that propogage information down the syntax tree. These attributes are declared using the `inherited` modifier instead of the `synthesized` one, but are othewise the same.

The file `BetterPP.sv` in the tutorial grammar `dc` defines a synthesized attribute `bpp` that unparses the abstract syntax tree without the use of unneccessary parenthesis. For example, for an expression such as `1 + 2 * 3`, the value of the `pp` attribute on the root of the abstract syntax tree is `(1 + (2 * 3))` while the value of `bpp` on the same node is `1 + 2 * 3`. To compute `bpp` a couple of inherited attributes are defined. One is `enclosingOpPrecedence` which indicates the operator predence of the operator on the enclosing parent of the node. Another, `leftOrRight`, indicates if an expression appears as the left or right child of the enclosing binary operator node. These values are used to determine if the `bpp` of the node should be wrapped in parenthesis or not.

While definitions of synthesized attributes compute the value of an attribute for the nonterminal node on the left hand side of a production, as in

```
e.pp = l.pp ++ " + " ++ r.pp ; ,
```

definitions of inherited attributes compute the value of an attribute for a nonterminal on the right hand side of a production. Four inherited attribute definitions appear in the following *aspect* of the abstract `add` production in `BetterPP.sv`:

```
aspect production add
sum::Expr ::= l::Expr r::Expr
{
 sum.bpp = if wrapInParens ( sum.enclosingOpPrecedence, 1,
                             sum.leftOrRight, "left" )
           then "(" ++ l.bpp ++ " + " ++ r.bpp ++ ")"
           else l.bpp ++ " + " ++ r.bpp ;

 l.enclosingOpPrecedence = 1 ;
 r.enclosingOpPrecedence = 1 ;
 l.leftOrRight = "left" ;
 r.leftOrRight = "right" ;
}
```

An *aspect* production simply provides a mechanism for defining additional attributes on an already existing production. The production `add` is defined in the file `AbstractSyntax.sv` and the aspect production above has the same effect as if the five attribute definitions were written on the `add` production in `AbstractSyntax.sv`. These inherited attribute definitions on `add` assign values for the inherited attributes of the children of the production.

Code generation: This phase of most language translators has no real analog in our simple expression language `dc`. A simple way to generate code is by using a string synthesized attribute to compute the translation. As an exercise the reader could define a synthesized attribute than generates the postfix version of the expresssion. In most programming languages other attributes that decorate the tree, such as the type of expressions, are used in performing the translation.

Code optimization is an important phase in compilation and can be performed using attribute grammars but is beyond the scope of this tutorial.

6 Running Silver and the generated language translators

Running Silver: Silver is run over a specified grammar and builds the language translator specified by that grammar. This will involve importing other grammars used by specified grammar and calling Copper to build any parsers needed by the grammar. The name of this grammar is provided one the command line when calling Silver.

Silver translates the attribute grammar specification to Java code that implements it. It also extracts the concrete syntax specifications that are to be used by Copper to generate the Java-based parsers specified in the concrete syntax of the grammar.

Since Silver is distributed as a Java jar file, this amounts to executing the jar. The tutorial grammar `dc` comes with a script `silver-compile` that shows a sample invocation of Silver.

This script also calls `ant`, a Java-based variant of the Unix `make` command, that compiles the Java code generated by Silver and invokes Copper to generate the requires parsers and scanners.

Running the generated langauge translators: Silver, like all attribute grammar systems, can be seen as specialized forms of lazy functional programming. Thus specifying side-effecting computations such as printing to the screen or writing to a file present some challenges.

The current version of Silver has adopted a model similar to that used in Haskell before Haskell was extended with monads. Essentially an input/output token is passed into the main function `main` and passed between side-effecting computations to control the order in which they are evaluated. This is certainly the most unappealing aspect to Silver and new users are not expected to dive into the details of this.

As a simple example, consider `Main.sv` in `dc`. The `main` function takes as input the command line arguments and the initial I/O token. Since the command line contains the string to be parsed this is passed to the function `parse`. This token is used by the calls to `print` in local attributes `print_success` and `print_failure`. These are attributes of type `IO` that represent the I/O token after the print statement has been executed.

Since Silver is a lazy functional language, these attributes are not evaluated until they are needed. It is the `return` statement in `main` that determines if the parse was successful and then demands only one of the I/O values. This causes exactly one of the print statmentis to be executed.

The `Driver.sv` file in `simple:host:driver` shows the use of other side-effecting I/O functions for reading and writing to files.

New users of Silver are encouraged to copy and paste from specifications such as these until they get more experience with Silver and develop a better understanding of its admittedly odd I/O system.

Now that parametric polymorphism has been added to Silver we expect to develop a monadic extension to Silver shortly that will bring it inline with the spirit of Haskell's monadic I/O sytem.

7 What next?

After reading this tutortial reader will hopefully have a reasonable first impression of how Silver and Copper work together to generate language translators from high-level attribute grammar specifications.

Silver contains many other constructs that are useful in defining extensible and domain-specific languages such as forwarding [5], reference attributes [2], collection attributes [1], parametric polymorphism, and import and export statements for the modular specification and composition of langauges. These are described in more detail in the upcoming guide "Building exteinsible and domain-specific langauges using Silver and Copper." The guide assumes no knowledge of Silver or Copper, but does assume the basic understanding of scanners, parsers, and attribute grammars that this tutorial aims to provide.

Readers may also be interested in our papers published on these tools. One might start with our paper "Silver: an Extensible Attribute Grammar System" [4].

In this document we have stated that the scanner produces a sequence of tokens from the input text, implying that it does this without any other input. While this is true for traditional scanners and parsers such as those generated by Lex or Flex, it is not true for Copper which introduced a notion of context-aware scanning. The parser calls the scanner asking for the "next token" as it is needed by the parser. Thus these are computed on demand. Furthermore the scanner using contextual information provided by the parser to tell it what terminal symbols are valid at that current point in the program. It uses this information to be more discriminating in determining the token to be returned. The paper "Context aware scanning for parsing extensible languages" [6] provides a full description of this process and some examples illustrating where it is useful.

Further information about Silver and Copper, links to the papers mentioned above, as well as the most current release of the tools are available at `http://melt.cs.umn.edu`.

References

[1] J. T. Boyland. Remote attribute grammars. *J. ACM*, 52(4):627–687, 2005.

[2] G. Hedin. Reference attribute grammars. *Informatica*, 24(3):301–317, 2000.

[3] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. Corrections in **5**(1971) pp. 95–96.

[4] E. Van Wyk, D. Bodin, J. Gao, and L. Krishnan. Silver: an extensible attribute grammar system. *Science of Computer Programming*, 75(1–2):39–54, January 2010.

[5] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In *Proc. 11th Intl. Conf. on Compiler Construction*, volume 2304 of *LNCS*, pages 128–142, 2002.

[6] E. Van Wyk and A. Schwerdfeger. Context-aware scanning for parsing extensible languages. In *Intl. Conf. on Generative Programming and Component Engineering, (GPCE)*. ACM Press, October 2007.

[7] H. Vogt, S. D. Swierstra, and M. F. Kuiper. Higher-order attribute grammars. In *ACM Conf. on Programming Language Design and Implementation (PLDI)*, pages 131–145, 1990.