TDDE45 - Lecture 5: Domain-Specific Languages

Adrian Pop and Martin Sjölund

Department of Computer and Information Science
Linköping University

2023-09-20

Part I

Domain-Specific Languages (DSLs)

Domain-Specific Languages

- Many are similar to classic, general-purpose programming languages (e.g. PHP).
- Examples include Unix shells, SQL, HTML, regular expressions, parser generators, some XML schemas, and many more.
- Compilers are usually implemented partially using domain-specific languages (grammars, special languages to describe architectures, etc).
- Why? It is easier to program and maintain such code.

DSLs: Markup Languages

- Markdown (.md) - used on github, gitlab for example
- Wiki - various flavors - used on Wikipedia, Trac, etc
- Doxygen - generate documentation from source code
- Sphinx - generate documentation from reStructuredText (.rst)
- LaTeX (.tex) - used to write articles, books and such

```
\begin{frame}[fragile]{DSLs: Markup Languages}
\begin{itemize}
\item Markdown (.md) - used on github, gitlab for example
\item Wiki - various flavors - used on Wikipedia, Trac, etc
\item Doxygen - generate documentation from source code
\item Sphinx - generate documentation from reStructuredText (.rst)
\item LaTeX (.tex) - used to write articles, books and such
\end{itemize}
\end{frame}
```

DSLs: Markup Languages - HTML

```html
<!DOCTYPE html>
<html>
<body>

<h1>My first HTML page</h1>

<p>Hello, world!</p>

</body>
</html>
```

HTML is markup code (some of which is interpreted)

DSLs: Template Languages

```
<!DOCTYPE html>
<html>
<body>

<h1>My first PHP page</h1>
<?php
echo $_SERVER["REMOTE_ADDR"];
?>
</body>
</html>
```

PHP code (highlighted)

DSLs: Template Languages

```
<!DOCTYPE html>
<html>
<body>

<h1>My first PHP page</h1>
<?php
echo $_SERVER["REMOTE_ADDR"];
?>
</body>
</html>
```

PHP code in-between pieces of code or markup. Typical usage is in web services (Facebook uses their own language derived from PHP because PHP at the time was too slow; modern PHP is slightly faster than Facebook's HHVM).

DSLs: Embedded Scripting Languages

```html
<!DOCTYPE html>
<html>
<body>

<p id="demo"></p>

<script>
document.getElementById("demo").innerHTML = "Hello World!";
</script>

</body>
</html>
```

JavaScript (JS) is interpreted code that was fast enough to be embedded in web-browsers back in 1997 (but JS is more like a general-purpose language these days)

DSLs: Shell Scripting Languages

```
#!/bin/bash

if test -f testsuite/Makefile; then
  cd testsuite
  for test in *.test; do
    grep "status: *correct" "$test"
  done
fi
```

Bash is either an interactive shell or interpreted code suitable for running system commands

DSLs: Regular expressions

```
# Look for line starting with status: correct
grep "^status: *correct" "$test"
# Look for openmodelica.org in the apache2 config
grep -R "openmodelica[.]org" /etc/apache2
# Replace all occurrences of http with https in the file
sed -i s,http://,https://,g file
sed -i s/SearchedText/ReplacedText/g file
```

Regular expressions appear almost everywhere from text editors to the venerable `grep` or `sed`.

DSLs: Build configuration

```
AC_PREREQ([2.63])
AC_INIT([OMCompiler],[dev],[https://trac.openmodelica.org/OpenModelica],
  [openmodelica],[https://openmodelica.org])
AC_LANG([C])
AC_PROG_CC
AC_SEARCH_LIBS(dlopen,dl)
AC_SUBST(EXTRA_LDFLAGS)
# ...
AC_OUTPUT(Makefile)
```

`autoconf` (m4) translates a description of possible build configurations and generates a shell script (./configure) that configures for example `Makefile` files.

DSLs: Build systems - make

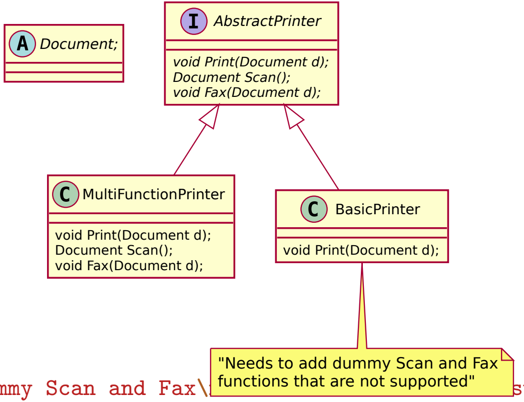
Example partial Makefile:

```
# Makefile.in
EXTRA_LDFLAGS=@EXTRA_LDFLAGS@

SomeFile.o: SomeCommand.c SomeCommand.h
	$(CC) -o $@ -c $< $(CFLAGS)

libSomeLib.so: $(DEPS) SomeFile.o
	@rm -f $@
	$(CC) -shared -o $@ $(DEPS) SomeFile.o $(LDFLAGS) $(EXTRA_LDFLAGS)
```

make interprets build dependencies and build rules to run shell commands

DSLs: Images and diagrams

```
@startuml
abstract class Document;
class MultiFunctionPrinter {
  void Print(Document d);
  Document Scan();
  void Fax(Document d);
}
class BasicPrinter {
  void Print(Document d);
}
interface AbstractPrinter {
  {abstract} void Print(Document d);
  {abstract} Document Scan();
  {abstract} void Fax(Document d);
}
note bottom of BasicPrinter : "Needs to add dummy Scan and Fax\
AbstractPrinter <|-- BasicPrinter
AbstractPrinter <|-- MultiFunctionPrinter
@enduml
```



PlantUML syntax for drawing a UML class diagram

DSLs: Parser Generators (Language Recognition)

```
(* a simple program syntax in EBNF − Wikipedia *)
program = 'PROGRAM', white_space, identifier, white_space,
          'BEGIN', white_space,
          { assignment, ";", white_space },
          'END.' ;
identifier = alphabetic_character, { alphabetic_character | digit } ;
number = [ "-" ], digit, { digit } ;
string = '"' , { all_characters - '"' }, '"' ;
assignment = identifier , ":=" , ( number | identifier | string ) ;
alphabetic_character = "A" | "B" | "C" | "D" | "E" | "F" | "G"
                     | "H" | "I" | "J" | "K" | "L" | "M" | "N"
                     | "O" | "P" | "Q" | "R" | "S" | "T" | "U"
                     | "V" | "W" | "X" | "Y" | "Z" ;
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
white_space = ? white_space characters ? ;
all_characters = ? all visible characters ? ;
```

See also courses in Formal Languages (TDDD14, etc) or Compiler Construction (TDDB44, TDDD55).

DSLs: Parser Generators (Language Recognition): Example

```
PROGRAM DEMO1
BEGIN
  A:=3;
  B:=45;
  H:=-100023;
  C:=A;
  D123:=B34A;
  BABOON:=GIRAFFE;
  TEXT:='Hello world!';
END.
```

Syntactically correct program according to the grammar on the previous slide.
Note that programs are usually parsed into an abstract syntax tree (the Composite design pattern).

DSLs: Special Purpose Language

```
sudoku6(Puzzle, Solution):-
  Solution = Puzzle,
  Puzzle = [S11, S12, S13, S14, S15, S16,
            S21, S22, S23, S24, S25, S26,
            S31, S32, S33, S34, S35, S36,
            S41, S42, S43, S44, S45, S46,
            S51, S52, S53, S54, S55, S56,
            S61, S62, S63, S64, S65, S66],
  fd_domain(Solution, 1, 6),
  Row1 = [S11, S12, S13, S14, S15, S16],
  Row2 = [S21, S22, S23, S24, S25, S26],
  Row3 = [S31, S32, S33, S34, S35, S36],
  Row4 = [S41, S42, S43, S44, S45, S46],
  Row5 = [S51, S52, S53, S54, S55, S56],
  Row6 = [S61, S62, S63, S64, S65, S66],
  Col1 = [S11, S21, S31, S41, S51, S61],
  Col2 = [S12, S22, S32, S42, S52, S62],
  Col3 = [S13, S23, S33, S43, S53, S63],
  Col4 = [S14, S24, S34, S44, S54, S64],
  Col5 = [S15, S25, S35, S45, S55, S65],
  Col6 = [S16, S26, S36, S46, S56, S66],
  Square1 = [S11, S12, S13, S21, S22, S23],
  Square2 = [S14, S15, S16, S24, S25, S26],
  Square3 = [S31, S32, S33, S41, S42, S43],
  Square4 = [S34, S35, S36, S44, S45, S46],
  Square5 = [S51, S52, S53, S61, S62, S63],
  Square6 = [S54, S55, S56, S64, S65, S66],

  valid([Row1, Row2, Row3, Row4, Row5, Row6,
         Col1, Col2, Col3, Col4, Col5, Col6,
         Square1, Square2, Square3, Square4, Square5, Square6]),
  writeRow(Row1),nl,
  writeRow(Row2),nl,nl,
  writeRow(Row3),nl,
  writeRow(Row4),nl,nl,
  writeRow(Row5),nl,
  writeRow(Row6),nl
  .

valid([]).
valid([Head | Tail]) :- fd_all_different(Head), valid(Tail).
writeRow(R) :-
  format('~d ~d ~d  ~d ~d ~d', R).

main :- sudoku6([_,_,_,1,_,6,6,_,4,
                 _,_,_,1,_,2,_,_,_,
                 _,_,_,5,_,1,_,_,_,
                 6,_,3,5,_,6,_,_,_], X), halt.

:- initialization(main).
```

Prolog program containing a Sudoku 6x6 solver. Declarative, no algorithm given.

Part II

Design with DSLs in mind

When you design software

- Would you write your own compiler?
  You try to use an existing programming language fulfilling all of your needs.
- Would you start by re-implementing your own standard library?
  You try to find a good library covering your needs.
- No good date parser in the standard library?
  Try to find a good third-party library covering your needs.
- Would you create your own library because nothing else fits and its useful in other projects?
  Maybe.

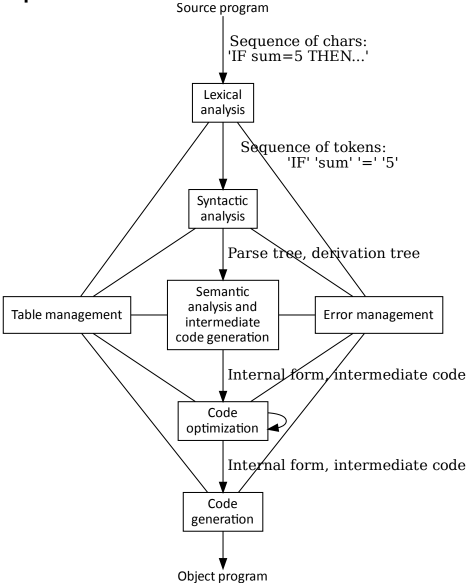
Design with DSLs in mind

- Would you write your own build system for your project?
  Re-use cmake or GNU autotools.
- Would you write your own image format for exporting a picture of your software?
  Generate postscript (for printing) or SVG.
- Would you write your own logic program or integer linear programming solver?
  Use an existing language and solver instead.
- Would you write your own help system?
  Re-use HTML renderers and write the help in HTML (or something that generates HTML) instead.
- Need to search text for some moderately fancy pattern?
  Regular expressions.
- Would you design your own language because nothing else fits?
  Possibly. Do you know compiler construction?

Part III

So how do you design a compiler or language?

The Phases of the Compiler



Example DSL: Modelica

- An equation-based object-oriented modeling language (a DSL).
- Modeling using a graphical user interface (or the equivalent textual representation).
- Used for simulation and/or control of multi-domain (physical) systems.
- Centered around making it easy for a (mechanical, electrical, etc) engineer to use Modelica.

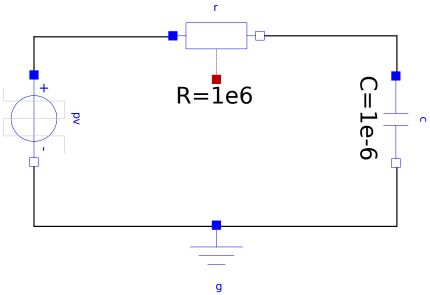
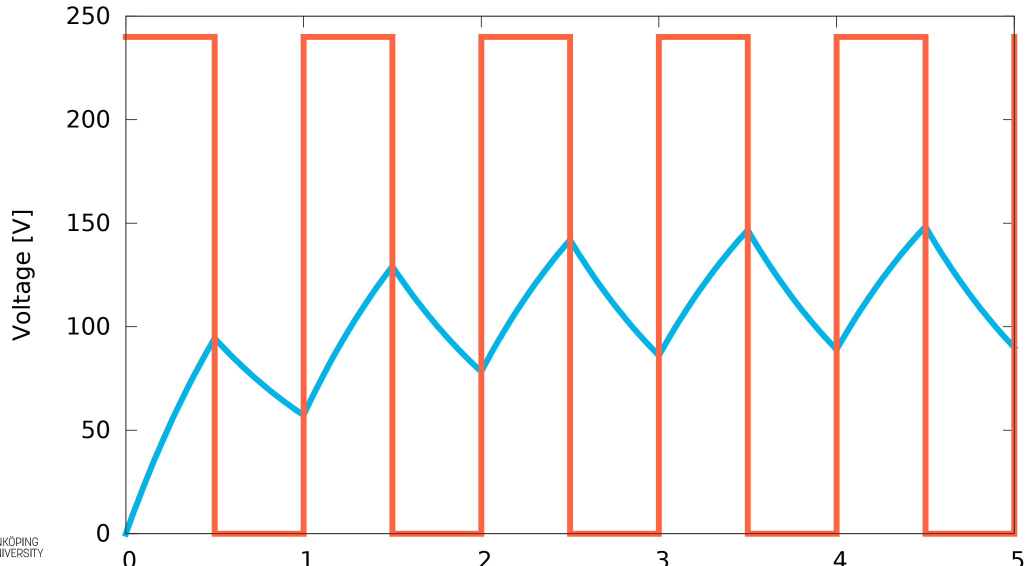


Figure: An RC-circuit constructed in Modelica by dragging-and-dropping components and connecting them.

Simulating the RC-circuit



Equations

Physics is described by *equations*, not statements. Thus, Modelica primarily uses equations instead of imperative programming (like C).

- Equations look like $\frac{V}{R} = I$

However, the declarative Modelica code needs to be translated into imperative programming (or similar) in order to run numerical solvers on a CPU. So it could be solved as either of:

- $V := R * I$
- $I := \frac{V}{R}$
- $R := \frac{V}{I}$

OpenModelica Parts

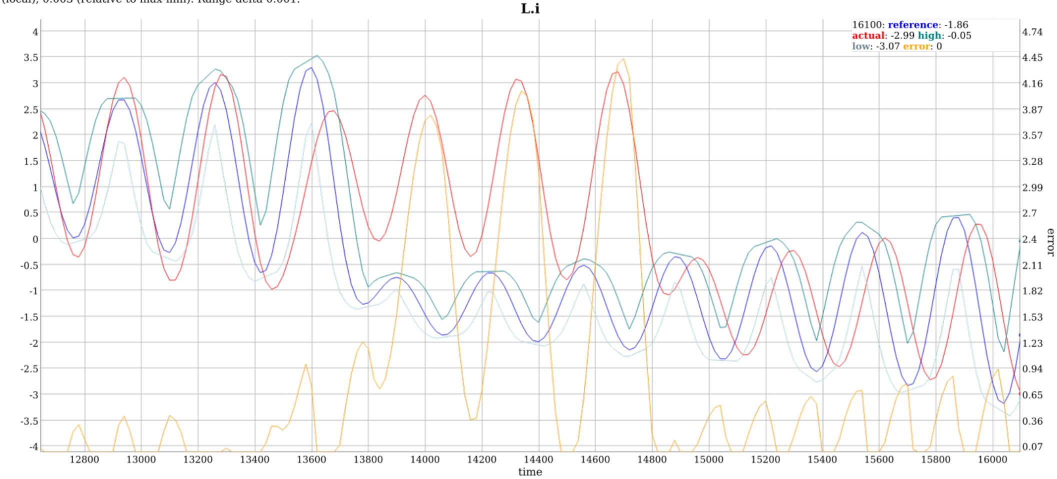
- Parser (using the ANTLR parser generator).
- Front-end (semantic analysis, like a traditional compiler).
- Equation back-end (symbolic math, outputs imperative code from equations).
- Code generator (takes the causal imperative code and generates C-code, skipping the middle-end and the back-end of a traditional compiler).
- Utilities.
- Scripting environment.
- Front-end + code generator handles MetaModelica (functions).
- The compiler is also written in MetaModelica (bootstrapping).

Testing a Modelica Compiler

- Testing a C-compiler is easier because the exact translation semantics are specified.
- In Modelica, a compiler needs to decide by itself how to generate code.
- Numerical differences depending on how an equation is solved.
- Compare result-files with a relative + absolute tolerance and some magic to align discrete event times.

Testing a Modelica Compiler

☑ reference ☑ actual ☑ high ☑ low ☑ error ☐ actual (original) Parameters used for the comparison: Relative tolerance 0.003 (local), 0.003 (relative to max-min). Range delta 0.001.



Next

- Seminar on cross platform on Friday
- DSL lab on Monday

www.liu.se