

DHBW MANNHEIM

STUDENT RESEARCH PROJECT

TINF11AI-BC

Implementing a SetlX to Python compiler

Author:

Jan-Christoph KLIE

Supervisor:

Prof. Dr. Karl STROETMANN

May 29, 2014

Contents

I	User Guide	2
1	Preface	3
1.1	Overview	3
1.2	Dependencies	3
2	Windows	4
2.1	Install Python	4
2.1.1	Obtain the Python binary	4
2.1.2	Setting the \$PATH	4
2.2	Install package manager	4
2.3	Install binary dependencies	5
2.4	Install Python dependencies	6
3	Usage	7
II	The SetlX language	8
4	Lexical analysis	9
4.1	Line structure	9
4.1.1	Logical lines	9
4.1.2	Physical lines	9
4.1.3	Comments	9
4.1.4	Blank lines	9
4.1.5	Whitespace	9
4.2	Identifiers and keywords	9
4.2.1	Identifier	10
4.2.2	Functor	10
4.3	Keywords	10
4.4	Literals	10
4.4.1	Character sequences	10
4.4.2	Numeric literals	11
4.5	Operators	12
4.6	Delimiters	12
5	Grammar	13
5.1	Top-level components	13
5.2	Expressions	14
5.2.1	Atom	15
5.2.2	Primaries	16
5.2.3	Enclosures	17
5.3	Simple statements	19
5.3.1	Assert statement	19
5.3.2	Assignment statement	19
5.3.3	Augmented assignment statement	19
5.3.4	Backtrack statement	19
5.3.5	Break statement	20
5.3.6	Continue statement	20
5.3.7	Exit statement	20

5.3.8	Expression statement	20
5.3.9	Return statement	20
5.4	Compound statements	21
5.4.1	Check statement	21
5.4.2	Class statement	21
5.4.3	Do-While statement	21
5.4.4	For-Loop statement	21
5.4.5	If statement	22
5.4.6	Match statement	22
5.4.7	Scan statement	22
5.4.8	Switch statement	22
5.4.9	Try statement	23
5.4.10	While-Loop statement	23
III	Setlx2py internals	25
6	Project structure	26
6.1	Source code	27
6.1.1	setlx_cfg	27
6.1.2	setlx_ast	27
6.1.3	setlx_transformer	27
6.1.4	setlx_builtin	27
6.1.5	setlx_codegen	27
6.1.6	setlx_lexer	27
6.1.7	setlx_parser	27
6.1.8	setlx_semcheck	28
6.1.9	setlx_util	28
6.2	Test files	29
6.2.1	test_builtin	29
6.2.2	test_execution	29
6.2.3	test_list & test_set	29
6.2.4	test_parsable	29
7	Testing	30
7.1	Test-driven development	30
7.2	Running the tests	31
8	AST	32
8.1	Configuration	32
8.2	Generation	32
9	Code generation	33
10	Transformation	34
10.1	Procedure definitions	34
10.2	Interpolation	35
11	Known issues	37

List of Figures

1	Running python from a shell without fully specified path	5
2	Installing the package manager pip	5
3	Installing the dependencies of setlx2py with pip	6
4	Using setlx2py to compile SetlX and running the generated Python	7
5	Folders and files in setlx2py	26
6	Output of nosetest: All tests passed	31

List of Abbreviations

AST Abstract Syntax Tree.

BNF Backus-Naur Form.

LALR Look-Ahead Left to Right, Rightmost derivation parser.

Part I

User Guide

1 Preface

1.1 Overview

Setlx2py consists of two parts: It is a compiler written in the Python programming language to transcompile SetlX sources to Python. In addition to that, it contains the needed runtime libraries. Therefore, when running code generated by setlx2py, an installed setlx2py environment is needed. When the development of setlx2py comes to a stable, binaries for the different target architectures are created simplify the step of installation.

In the following chapters, the steps needed to setup setlx2py on your system are described. It can be run on nearly any platform which has a standard Python implementation. At first, the general dependencies are described. After that, some in-depth guides for installing on Windows, Ubuntu Linux and Mac OS can be found. In the end, the actual usage of setlx2py is briefly explained.

1.2 Dependencies

To deal with a target platform which is not described here, or this installation guide becomes outdated, the dependencies necessary to run setlx2py are now explained.

Runtime Setlx2py is written in Python, and generates Python source code from SetlX input. Therefore, a Python runtime is needed. Any version starting from Python 2.7 is officially supported. It tested with version numbers 2.7 and 3.3. Older versions may work, but are not recommended nor supported.

Packages

- `PLY` ¹
- `blist` ²
- `nose` (only for testing) ³
- `ast-gen` (only for development) ⁴

¹<https://pypi.python.org/pypi/ply>

²<https://pypi.python.org/pypi/blist>

³<https://pypi.python.org/pypi/nose>

⁴<https://github.com/Rentier/ast-gen>

2 Windows

2.1 Install Python

2.1.1 Obtain the Python binary

First, download a Python installer from the official Website, which can be found at <https://www.python.org/download/>. Be sure to select an installer, not source packages. It should not matter which version you use, but recommended is the either the newest version of 2.7 or the newest version of 3.X. It is a matter of taste, and might depend on which version you might already have installed. But `setlx2py` should run with all newer versions of standard Python.

The setup itself is self-explanatory, just run the executable as usual. When you decide where to install, it is recommended to let Python install where it wants. If you choose a different installation path, remember where it was, you will need that information in the next step. The location of the Python installation is called `$PYHOME` for further reference.

2.1.2 Setting the \$PATH

Now you could run Python with the command

```
$ $PYHOME\python.exe
```

Typing the full path is very tedious. To ease the usage of the Python interpreter, Windows has the feature of the `$PATH`-Variable. Whenever you insert a command without a fully specified path, it looks in the folders you specified in that variable, and looks for a match. There are two ways to add a program to the path variable. Open a Powershell with either `Win` + `R` + `'powershell'` + `Enter`. Insert the following command in the just opened window:

```
$ [Environment]::SetEnvironmentVariable("Path", "$env:Path;C:\Python27\;C:\Python27\Scripts\","User")
```

In that example, we used Python 2.7. If you installed another version of python, or used a different `$PYHOME`, adjust the part of `C:\Python27` to reflect the differences.

Close the shell and open it again. You should now be able to open a Python prompt with entering

```
$ python
```

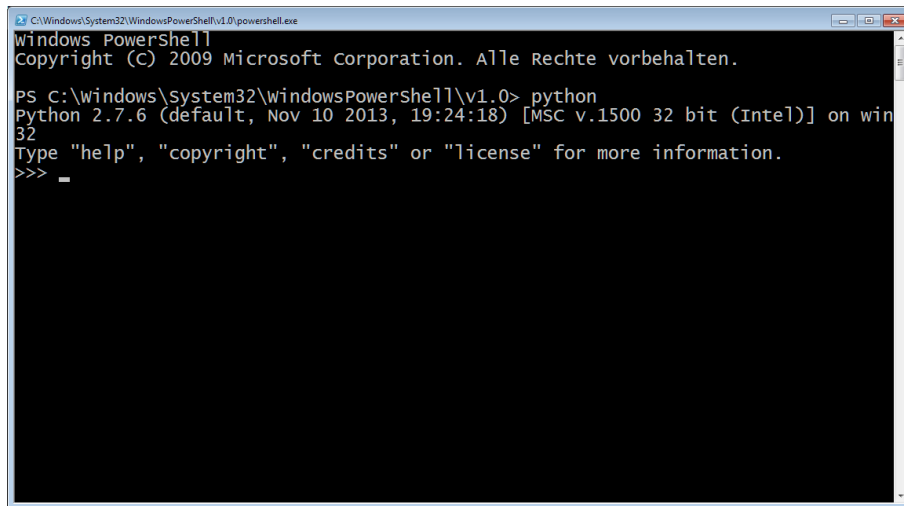
in the shell.

2.2 Install package manager

Python allows installing packages in a very easy way. Sadly, it does not ship with a package manager. Therefore, it is installed in the next step. Download

```
https://raw.githubusercontent.com/pypa/pip/master/contrib/get-pip.py
```

Open a shell and change the directory to the folder to where you downloaded it (the `cd` + `$FOLDERNAME` command does that for you). Now run



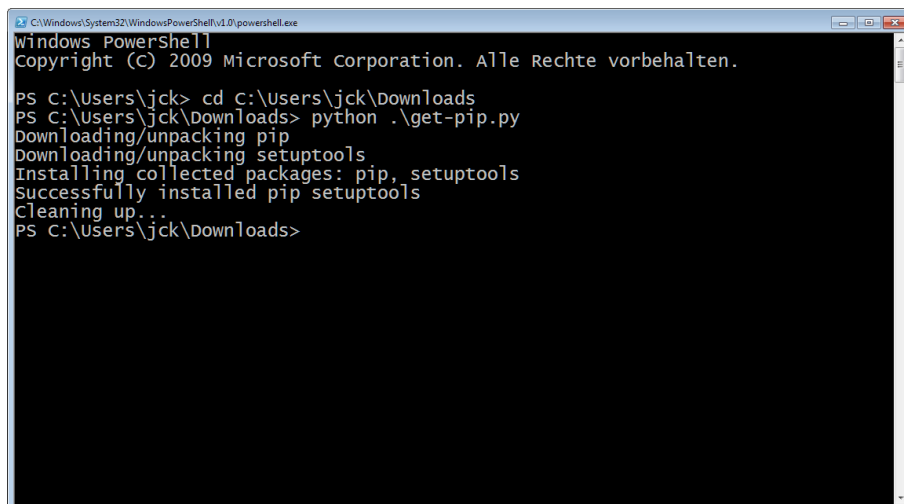
```
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. Alle Rechte vorbehalten.

PS C:\Windows\System32\WindowsPowerShell\v1.0> python
Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit (Intel)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

Figure 1: Running python from a shell without fully specified path

```
$ python \.get-pip.py
```

You should see a confirmation that it is installed everything successfully like in Fig. 2.



```
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. Alle Rechte vorbehalten.

PS C:\Users\jck> cd c:\Users\jck\Downloads
PS C:\Users\jck\Downloads> python .\get-pip.py
Downloading/unpacking pip
Downloading/unpacking setuptools
Installing collected packages: pip, setuptools
Successfully installed pip setuptools
Cleaning up...
PS C:\Users\jck\Downloads>
```

Figure 2: Installing the package manager pip

2.3 Install binary dependencies

Setlx2py uses internally some platform-specific binary libraries. It is possible to compile these for Windows with some effort, but they can be found precompiled already.

Download the fitting version of the **blist** package (for the curious, it offers better/more advanced data structures) and install it. Be sure to download the same architecture as you did for Python (32- or 64-bit), else you get an error with a name of ‘Cannot install’ or similar. The package itself can be obtained from

<http://www.lfd.uci.edu/~gohlke/pythonlibs/#blist>

2.4 Install Python dependencies

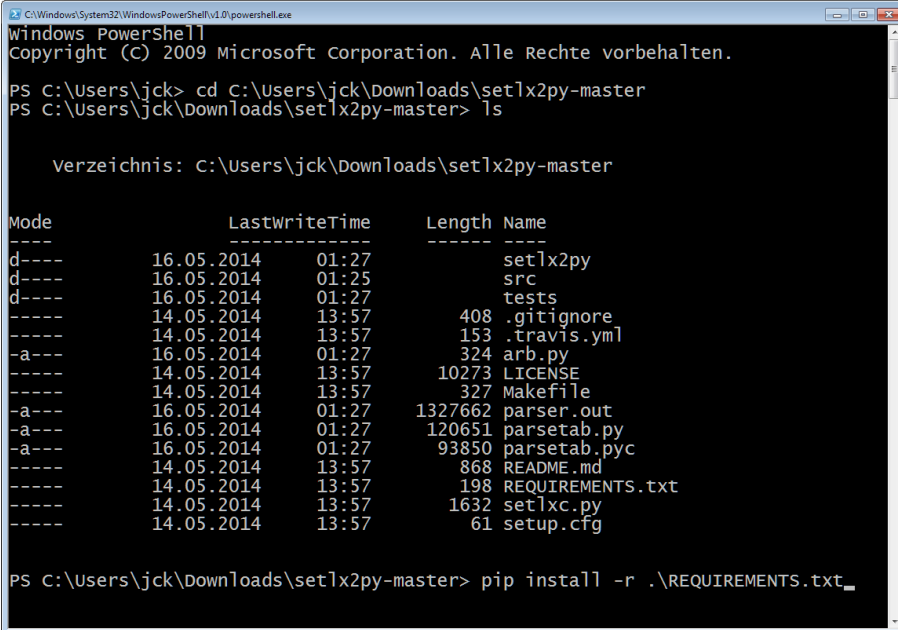
Download the setlx2py archive from

<https://github.com/Rentier/setlx2py/archive/master.zip>

and extract it. Open a shell and change the working directory to the folder where the REQUIREMENTS.txt file can be found. With the power of pip, all other dependencies can now be installed with

```
$ pip install -r REQUIREMENTS.txt
```

The package manager now retrieves all the dependencies specified in the given file. After some time, it messages 'Successfully installed'. Everything is now in place to actually use setlx2py.



```
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. Alle Rechte vorbehalten.

PS C:\Users\jck> cd C:\Users\jck\Downloads\setlx2py-master
PS C:\Users\jck\Downloads\setlx2py-master> ls

Verzeichnis: C:\Users\jck\Downloads\setlx2py-master

Mode                LastWriteTime         Length Name
----                -
d-----         16.05.2014         01:27      setlx2py
d-----         16.05.2014         01:25         src
d-----         16.05.2014         01:27      tests
-----         14.05.2014         13:57         408 .gitignore
-----         14.05.2014         13:57         153 .travis.yml
-a----         16.05.2014         01:27         324 arb.py
-----         14.05.2014         13:57       10273 LICENSE
-----         14.05.2014         13:57         327 Makefile
-a----         16.05.2014         01:27     1327662 parser.out
-a----         16.05.2014         01:27     120651  parsetab.py
-a----         16.05.2014         01:27     93850  parsetab.pyc
-----         14.05.2014         13:57         868 README.md
-----         14.05.2014         13:57         198 REQUIREMENTS.txt
-----         14.05.2014         13:57        1632 setlxc.py
-----         14.05.2014         13:57          61 setup.cfg

PS C:\Users\jck\Downloads\setlx2py-master> pip install -r .\REQUIREMENTS.txt_
```

Figure 3: Installing the dependencies of setlx2py with pip

3 Usage

Using setlx2py consists of two steps. At first, one has to compile a given SetlX source file. That can be done with

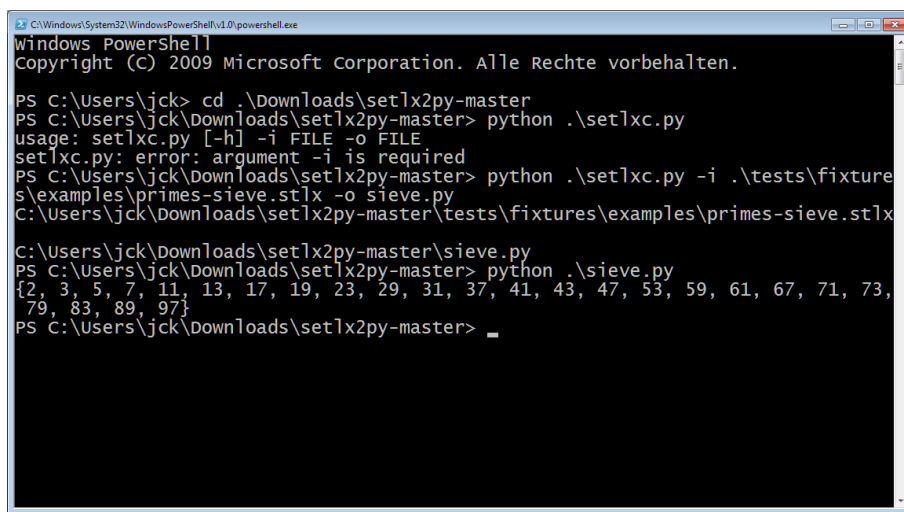
```
$ setlxc.py -i STLX_IN -o PY_OUT
```

where STLX_IN is the path to the SetlX source file, and PY_OUT the path to the file where the Python code will be saved.

The generated file then has to be run with the Python interpreter:

```
$ python PY_OUT
```

The complete steps can be seen in the Fig. 4.



```
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. Alle Rechte vorbehalten.

PS C:\Users\jck> cd .\Downloads\setlx2py-master
PS C:\Users\jck\Downloads\setlx2py-master> python .\setlxc.py
usage: setlxc.py [-h] -i FILE -o FILE
setlxc.py: error: argument -i is required
PS C:\Users\jck\Downloads\setlx2py-master> python .\setlxc.py -i .\tests\fixtures\examples\primes-sieve.stlx -o sieve.py
C:\Users\jck\Downloads\setlx2py-master\tests\fixtures\examples\primes-sieve.stlx
C:\Users\jck\Downloads\setlx2py-master\sieve.py
PS C:\Users\jck\Downloads\setlx2py-master> python .\sieve.py
{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97}
PS C:\Users\jck\Downloads\setlx2py-master> _
```

Figure 4: Using setlx2py to compile SetlX and running the generated Python

Part II

The SetlX language

4 Lexical analysis

The following chapter describes the tokens used in setlx2py. The model used is heavily borrowed from the one Python 2.7. It can be found in [3].

4.1 Line structure

A SetlX program is built of logical lines.

4.1.1 Logical lines

The end of a logical line is represented by a semicolon ‘;’.

4.1.2 Physical lines

A physical line consists of a number of characters ending with a end-of-line character. One physical line can have any fraction of a logical line, i.e. one can split code arbitrarily at whitespace.

4.1.3 Comments

Setlx has two types of comments. Comments are ignored by the lexer, they are stripped and do not generate any tokens.

Singe-line comments

A single line comment starts with a double slash (//) that is not part of a string literal, and ends at the end of the physical line.

Multi-line comments

Multi-line comments, start with /* and end with */.

4.1.4 Blank lines

Lines that only contain whitespace are ignored.

4.1.5 Whitespace

Except in string literals, the whitespace characters used to separate tokens are tab, carriage return, form feed, vertical tab.

4.2 Identifiers and keywords

Identifier and functors are unlimited in length. Case is significant.

4.2.1 Identifier

Identifiers are described by the following lexical definitions:

$\langle identifier \rangle$	$::= \langle lowercase \rangle \langle seq \rangle \mid \text{'_}'$
$\langle seq \rangle$	$::= \langle seq \rangle \langle char \rangle$ $\mid \epsilon$
$\langle char \rangle$	$::= \langle lowercase \rangle \mid \langle uppercase \rangle \mid \langle digit \rangle \mid \text{'_}'$
$\langle lowercase \rangle$	$::= \text{'a'} \dots \text{'z'}$
$\langle uppercase \rangle$	$::= \text{'A'} \dots \text{'Z'}$
$\langle digit \rangle$	$::= \text{'0'} \dots \text{'9'}$

4.2.2 Functor

Functors are described by the following lexical definitions:

$\langle functor \rangle$	$::= \langle uppercase \rangle \langle seq \rangle$
---------------------------	---

4.3 Keywords

The following identifier are treated as keywords in SetlX and cannot used for other purposes:

true	false	in	notin	forall	exists
backtrack	check	match	regex	as	break
continue	exit	return	assert	if	else
switch	case	default	for	do	while
procedure	cachedProcedure	class	static	scan	using
try	catch	catchUsr	catchLng		

4.4 Literals

Literals are notations for constant values of some built-in types.

4.4.1 Character sequences

Strings

Any sequence of characters enclosed in double quotes which does not contain a interpolation is considered a string.

Literal strings

Any sequence of characters enclosed in single quotes is considered a literal string.

String interpolations

A string interpolation is a string which contains at least one interpolation.

$$\langle interpolation \rangle ::= \$ \langle expression \rangle \$$$

Escaping

SetlX supports the same escape sequences as the language C.

4.4.2 Numeric literals

There are two types of numeric literals: integer and floating point numbers. Note that the following definitions do not include signs, that is handled by expressions with unary operators.

Integer literals

$$\langle integer \rangle ::= \langle nonzerodigit \rangle \langle digits-or-empty \rangle \mid '0'$$
$$\langle nonzerodigit \rangle ::= '1' \dots '9'$$
$$\begin{aligned} \langle digits \rangle &::= \langle digits \rangle \langle digit \rangle \\ &\mid \langle digit \rangle \end{aligned}$$
$$\langle digits-or-empty \rangle ::= \langle digits \rangle \mid \epsilon$$

Floating point literals

Floating point literals are described by the following lexical definitions:

$$\langle floatnumber \rangle ::= \langle pointfloat \rangle \mid \langle exponentfloat \rangle$$
$$\begin{aligned} \langle pointfloat \rangle &::= \langle fraction \rangle \\ &\mid \langle intpart \rangle \langle fraction \rangle \\ &\mid \langle intpart \rangle '.' \end{aligned}$$
$$\langle exponentfloat \rangle ::= \langle significand \rangle \langle exponent \rangle$$
$$\langle significand \rangle ::= \langle intpart \rangle \mid \langle pointfloat \rangle$$
$$\langle intpart \rangle ::= \langle digits \rangle$$
$$\langle fraction \rangle ::= '.' \langle digits \rangle$$
$$\langle exponent \rangle ::= \langle e \rangle \langle sign \rangle \langle digits \rangle$$

$\langle e \rangle ::= 'e' \mid 'E'$

$\langle sign \rangle ::= '+' \mid '-' \mid \epsilon$

4.5 Operators

The following tokens are operators:

+	-	/	*	\	%
><	**	#	@		
<==>	<!=>	=>		&&	!
==	!=	<	<=	>	>=
\+	*				

4.6 Delimiters

The following tokens serve as delimiters in the grammar:

()	[]	{	}
;	,	:	..	.	
:=	+=	-=	*=		
/=	\=	%=	->		

5 Grammar

In this chapter, the grammar of SetlX is explained. The syntax used to describe it is BNF. The grammar was implemented in a LALR(1) parser, therefore, the grammar is LALR(1).

In some cases, the reader might think of an easier way to express given constructs. These strange and too complex looking rules are in most cases written that way to preserve the LALR nature and to deal with parsers which do not offer operator precedence.

The grammar used is heavily borrowed from the Python 2.7 grammar. It can be found in [3].

5.1 Top-level components

All input read from files has the same form:

$\langle S \rangle$	$::= \langle file_input \rangle$
$\langle file_input \rangle$	$::= \langle statement_list \rangle$ $\langle expression \rangle$
$\langle statement_list \rangle$	$::= \langle statement \rangle$ $\langle statement_list \rangle \langle statement \rangle$
$\langle statement \rangle$	$::= \langle simple_statement \rangle ';' $ $\langle compound_statement \rangle$
$\langle block \rangle$	$::= \langle statement_list \rangle$ ϵ

5.2 Expressions

This section explains the elements occurring in SetIX expressions.

$\langle expression \rangle$	$::=$	$\langle implication \rangle$ $\langle lambda_definition \rangle$ $\langle implication \rangle$ ‘ \Leftarrow ’ $\langle implication \rangle$ $\langle implication \rangle$ ‘ $\Leftarrow!$ ’ $\langle implication \rangle$
$\langle implication \rangle$	$::=$	$\langle disjunction \rangle$ $\langle disjunction \rangle$ ‘ \Rightarrow ’ $\langle disjunction \rangle$
$\langle disjunction \rangle$	$::=$	$\langle conjunction \rangle$ $\langle disjunction \rangle$ ‘ \vee ’ $\langle conjunction \rangle$
$\langle conjunction \rangle$	$::=$	$\langle comparison \rangle$ $\langle conjunction \rangle$ ‘ $\&\&$ ’ $\langle comparison \rangle$
$\langle comparison \rangle$	$::=$	$\langle sum \rangle$ $\langle sum \rangle$ ‘ $=$ ’ $\langle sum \rangle$ $\langle sum \rangle$ ‘ $!=$ ’ $\langle sum \rangle$ $\langle sum \rangle$ ‘ $<$ ’ $\langle sum \rangle$ $\langle sum \rangle$ ‘ $<=$ ’ $\langle sum \rangle$ $\langle sum \rangle$ ‘ $>$ ’ $\langle sum \rangle$ $\langle sum \rangle$ ‘ $>=$ ’ $\langle sum \rangle$ $\langle sum \rangle$ ‘ in ’ $\langle sum \rangle$ $\langle sum \rangle$ ‘ notin ’ $\langle sum \rangle$
$\langle sum \rangle$	$::=$	$\langle product \rangle$ $\langle sum \rangle$ ‘ $+$ ’ $\langle product \rangle$ $\langle sum \rangle$ ‘ $-$ ’ $\langle product \rangle$
$\langle product \rangle$	$::=$	$\langle reduce \rangle$ $\langle product \rangle$ ‘ $*$ ’ $\langle reduce \rangle$ $\langle product \rangle$ ‘ $/$ ’ $\langle reduce \rangle$ $\langle product \rangle$ ‘ \backslash ’ $\langle reduce \rangle$ $\langle product \rangle$ ‘ $\%$ ’ $\langle reduce \rangle$ $\langle product \rangle$ ‘ $<>$ ’ $\langle reduce \rangle$
$\langle reduce \rangle$	$::=$	$\langle unary_expression \rangle$ ‘ $\backslash+$ ’ $\langle unary_expression \rangle$ ‘ $\backslash*$ ’ $\langle unary_expression \rangle$
$\langle unary_expression \rangle$	$::=$	$\langle power \rangle$ ‘ $\backslash+$ ’ $\langle unary_expression \rangle$ ‘ $\backslash*$ ’ $\langle unary_expression \rangle$ ‘ $\#$ ’ $\langle unary_expression \rangle$ ‘ $-$ ’ $\langle unary_expression \rangle$ ‘ $@$ ’ $\langle unary_expression \rangle$

	'!' $\langle unary_expression \rangle$ $\langle quantor \rangle$ $\langle term \rangle$
$\langle power \rangle$	$::= \langle primary \rangle$ $\langle primary \rangle '**' \langle power \rangle$
$\langle primary \rangle$	$::= \langle atom \rangle$ $\langle attributeref \rangle$ $\langle subscription \rangle$ $\langle slicing \rangle$ $\langle procedure \rangle$ $\langle call \rangle$ $\langle primary \rangle '!'$
lambda	
$\langle lambda_definition \rangle$	$::= \langle lambda_parameters \rangle ' ->' \langle expression \rangle$
$\langle lambda_parameters \rangle$	$::= \langle identifier \rangle$ $\langle list_display \rangle$
quantor	
$\langle quantor \rangle$	$::= 'forall' '(' \langle iterator_chain \rangle ' ' \langle expression \rangle ')'$ $'exists' '(' \langle iterator_chain \rangle ' ' \langle expression \rangle ')'$
term	
$\langle term \rangle$	$::= 'TERM' '(' \langle argument_list \rangle ')'$ $'TERM' '(' ')'$

5.2.1 Atom

Atoms (as the name implies) are the fundamental elements of expressions.

$\langle atom \rangle$	$::= \langle identifier \rangle$ $\langle literal \rangle$ $\langle enclosure \rangle$
$\langle identifier \rangle$	$::= 'IDENTIFIER'$ $'_'$

Literals

SetIX supports three kinds of string literals and two numeric literals:

$\langle literal \rangle$	$::= \langle stringliteral \rangle$ $\langle integer \rangle$
---------------------------	--

	$\langle floatnumber \rangle$
	$\langle boolean \rangle$
$\langle stringliteral \rangle$::= 'STRING'
	'LITERAL'
	'INTERPOLATION'
$\langle integer \rangle$::= 'INTEGER'
$\langle floatnumber \rangle$::= 'DOUBLE'
$\langle boolean \rangle$::= 'true'
	'false'

5.2.2 Primaries

Primaries represent the most tightly bound operations of the language. That is why they are at the bottom of the expression grammar.

Attributeref

An attribute reference is a primary followed by a period and a name:

$\langle attributeref \rangle ::= \langle primary \rangle \text{ '.' } \langle identifier \rangle$

Subscription

A subscription retrieves an item of an indexable:

$\langle subscription \rangle ::= \langle primary \rangle \text{ '[' } \langle expression \rangle \text{ ']'}$

Slicing

A slice retrieves a subset of the sliced object. SetlX allows leaving out either the lower or upper bound. The object sliced has to support the operation.

$\langle slicing \rangle ::= \langle primary \rangle \text{ '[' } \langle lower_bound \rangle \text{ '...' } \langle upper_bound \rangle \text{ ']'}$

$\langle lower_bound \rangle ::= \langle expression \rangle$
| ϵ

$\langle upper_bound \rangle ::= \langle expression \rangle$
| ϵ

Procedure

The definition of a function in SetlX is an expression, not a compound statement, since it needs to be bound to a variable to be usable.

There are two kinds of procedures: vanilla and cached.

$$\begin{aligned}
\langle procedure \rangle & ::= \text{'procedure' '(' parameter_list ')'} \text{'{' 'block' '}' } \\
& \quad | \text{'cProcedure' '(' parameter_list ')'} \text{'{' 'block' '}' } \\
\langle parameter_list \rangle & ::= \langle params \rangle \\
& \quad | \epsilon \\
\langle params \rangle & ::= \langle procedure_param \rangle \\
& \quad | \langle params \rangle \text{' ,' } \langle procedure_param \rangle \\
\langle procedure_param \rangle & ::= \langle identifier \rangle
\end{aligned}$$

Call

A call invokes a callable with a possible empty list of arguments.

$$\begin{aligned}
\langle call \rangle & ::= \langle primary \rangle \text{'(' } \langle argument_list \rangle \text{')'} \\
& \quad | \langle primary \rangle \text{'(' ')'} \\
\langle argument_list \rangle & ::= \langle expression \rangle \\
& \quad | \langle argument_list \rangle \text{' ,' } \langle expression \rangle
\end{aligned}$$

5.2.3 Enclosures

SetlX extensively uses enclosures for the syntax sugar of builtin data types.

$$\begin{aligned}
\langle enclosure \rangle & ::= \langle parenth_form \rangle \\
& \quad | \langle set_range \rangle \\
& \quad | \langle list_range \rangle \\
& \quad | \langle set_display \rangle \\
& \quad | \langle list_display \rangle \\
& \quad | \langle set_comprehension \rangle \\
& \quad | \langle list_comprehension \rangle
\end{aligned}$$

Parenthesized forms

A parenthesized form is an expression enclosed in parentheses. The fact that it is at the bottom of the expression tree gives it the highest precedence in arithmetic expressions.

$$\langle parenth_form \rangle ::= \text{'(' } \langle expression \rangle \text{')'}$$

Comprehensions

Comprehensions provide a concise way to create new instances of that type based on an already existing collection.

$$\begin{aligned}
\langle set_comprehension \rangle & ::= \text{'{' } \langle expression \rangle \text{' :' } \langle iterator_chain \rangle \langle comprehension_condition \rangle \text{'}' } \\
& \quad \text{'}' } \\
\langle list_comprehension \rangle & ::= \text{'[' } \langle expression \rangle \text{' :' } \langle iterator_chain \rangle \langle comprehension_condition \rangle \text{']'} \\
& \quad \text{']'}
\end{aligned}$$

$$\langle \textit{comprehension_condition} \rangle ::= \text{'|'} \langle \textit{expression} \rangle \\ | \quad \epsilon$$

Ranges

Ranges provide a short syntax to create collections which contain all values from a given start to an end with an optional specified step size. The expressions must evaluate to integers.

$$\langle \textit{set_range} \rangle ::= \text{'{' } \langle \textit{expression} \rangle \text{'..' } \langle \textit{expression} \rangle \text{'}} \\ | \quad \text{'{' } \langle \textit{expression} \rangle \text{' ,' } \langle \textit{expression} \rangle \text{'..' } \langle \textit{expression} \rangle \text{'}}$$

$$\langle \textit{list_range} \rangle ::= \text{'[' } \langle \textit{expression} \rangle \text{'..' } \langle \textit{expression} \rangle \text{']'} \\ | \quad \text{'[' } \langle \textit{expression} \rangle \text{' ,' } \langle \textit{expression} \rangle \text{'..' } \langle \textit{expression} \rangle \text{']'}$$

Displays

Displays are the syntax sugar for lists and sets. Values of the collection to be are enclosed in either round or curly braces.

These are also used in case statements for representing the matchee collections.

$$\langle \textit{set_display} \rangle ::= \text{'{' } \langle \textit{expression} \rangle \text{'}} \\ | \quad \text{'{' } \langle \textit{expression} \rangle \text{' ,' } \textit{argument_list} \text{'}} \\ | \quad \text{'{' } \text{'}} \\ | \quad \text{'{' } \langle \textit{expression} \rangle \text{'|'} \langle \textit{expression} \rangle \text{'}} \\ | \quad \text{'{' } \langle \textit{expression} \rangle \text{' ,' } \textit{argument_list} \text{'|'} \langle \textit{expression} \rangle \text{'}}$$

$$\langle \textit{list_display} \rangle ::= \text{'[' } \langle \textit{expression} \rangle \text{']'} \\ | \quad \text{'[' } \langle \textit{expression} \rangle \text{' ,' } \textit{argument_list} \text{']'} \\ | \quad \text{'[' } \text{']'} \\ | \quad \text{'[' } \langle \textit{expression} \rangle \text{'|'} \langle \textit{expression} \rangle \text{']'} \\ | \quad \text{'[' } \langle \textit{expression} \rangle \text{' ,' } \textit{argument_list} \text{'|'} \langle \textit{expression} \rangle \text{']'}$$

5.3 Simple statements

Simple statements form a single logical line and therefore have to end with a semi-colon.

$$\begin{aligned}\langle simple_statement \rangle & ::= \langle assert_statement \rangle \\ & \quad | \langle assignment_statement \rangle \\ & \quad | \langle augmented_assign_statement \rangle \\ & \quad | \langle backtrack_statement \rangle \\ & \quad | \langle break_statement \rangle \\ & \quad | \langle continue_statement \rangle \\ & \quad | \langle exit_statement \rangle \\ & \quad | \langle expression_statement \rangle \\ & \quad | \langle return_statement \rangle\end{aligned}$$

5.3.1 Assert statement

An assert statement throws an exception when the condition does not match the given expectation. This is useful for sanity checks and debugging during runtime.

$$\langle assert_statement \rangle ::= \text{'assert' '(' } \langle expression \rangle \text{ 'COMMA' } \langle expression \rangle \text{ ')}'$$

5.3.2 Assignment statement

Assignment statements bind the right-hand side to the names of the left-hand side. The target has to be checked to be a valid assignable, since the LALR nature of the grammar prevents using more strict non-terminals like list-displays.

$$\langle assignment_statement \rangle ::= \langle target \rangle \text{' := ' } \langle expression \rangle$$
$$\langle target \rangle ::= \langle expression \rangle$$

5.3.3 Augmented assignment statement

Augmented assignment statements combine assignment with an binary operation. See the note for assignment statements.

$$\langle augmented_assign_statement \rangle ::= \langle augtarget \rangle \langle augop \rangle \langle expression \rangle$$
$$\begin{aligned}\langle augtarget \rangle & ::= \langle identifier \rangle \\ & \quad | \langle attributeref \rangle \\ & \quad | \langle subscription \rangle\end{aligned}$$
$$\langle augop \rangle ::= \text{' += ' } | \text{' -= ' } | \text{' *= ' } | \text{' /= ' } | \text{' \= ' } | \text{' %= '}$$

5.3.4 Backtrack statement

$$\langle backtrack_statement \rangle ::= \text{'backtrack'}$$

5.3.5 Break statement

$\langle break_statement \rangle ::= \text{'break'}$

5.3.6 Continue statement

$\langle continue_statement \rangle ::= \text{'continue'}$

5.3.7 Exit statement

$\langle exit_statement \rangle ::= \text{'exit'}$

5.3.8 Expression statement

An expression statement evaluates the given expression.

$\langle expression_statement \rangle ::= \langle expression \rangle$

5.3.9 Return statement

$\langle return_statement \rangle ::= \text{'return'}$
 $| \text{'return' } \langle expression \rangle$

5.4 Compound statements

Compound statements contain other statements; in general, they span more than one line and are not ended by a semicolon.

$$\begin{aligned}\langle \textit{compound_statement} \rangle &::= \langle \textit{check_statement} \rangle \\ &| \langle \textit{class} \rangle \\ &| \langle \textit{do_while_loop} \rangle \\ &| \langle \textit{for_loop} \rangle \\ &| \langle \textit{if_statement} \rangle \\ &| \langle \textit{match_statement} \rangle \\ &| \langle \textit{scan_statement} \rangle \\ &| \langle \textit{switch_statement} \rangle \\ &| \langle \textit{try_statement} \rangle \\ &| \langle \textit{while_loop} \rangle\end{aligned}$$

5.4.1 Check statement

Belongs to backtracking.

$$\langle \textit{check_statement} \rangle \quad ::= \text{'check' '{' } \langle \textit{block} \rangle \text{'}'}$$

5.4.2 Class statement

$$\langle \textit{class} \rangle \quad ::= \text{'class' } \langle \textit{identifier} \rangle \text{'(' } \langle \textit{parameter_list} \rangle \text{'')' '{' } \langle \textit{block} \rangle \langle \textit{static_block} \rangle \text{'}'}$$
$$\begin{aligned}\langle \textit{static_block} \rangle &::= \text{'static' '{' } \langle \textit{block} \rangle \text{'}' } \\ &| \epsilon\end{aligned}$$

5.4.3 Do-While statement

Typical do-while loop.

$$\langle \textit{do_while_loop} \rangle \quad ::= \text{'do' '{' } \langle \textit{block} \rangle \text{'}' } \text{'while' '(' } \langle \textit{expression} \rangle \text{'')' ';'}$$

5.4.4 For-Loop statement

The for-loop iterates over the cartesian of the iterator chain, not the zip.

$$\langle \textit{for_loop} \rangle \quad ::= \text{'for' '(' } \langle \textit{iterator_chain} \rangle \text{'')' '{' } \langle \textit{block} \rangle \text{'}'}$$
$$\langle \textit{iterator} \rangle \quad ::= \langle \textit{comparison} \rangle$$
$$\begin{aligned}\langle \textit{iterator_chain} \rangle &::= \langle \textit{iterator} \rangle \\ &| \langle \textit{iterator_chain} \rangle \text{' ,' } \langle \textit{iterator} \rangle\end{aligned}$$

5.4.5 If statement

Typical if-else statement. Note that the block attached needs parenthesis, there is nothing like a single line body.

$$\begin{aligned}\langle if_statement \rangle & ::= 'if' '(' \langle expression \rangle ')' \{ \langle block \rangle \} \\ & \quad | 'if' '(' \langle expression \rangle ')' \{ \langle block \rangle \} 'else' \{ \langle block \rangle \} \\ & \quad | 'if' '(' \langle expression \rangle ')' \{ \langle block \rangle \} 'else' \langle if_statement \rangle\end{aligned}$$

5.4.6 Match statement

The match statement looks whether a matchee matches a given list on patterns and conditions, and then binds it according to the match.

$$\langle match_statement \rangle ::= 'match' '(' \langle expression \rangle ')' \{ \langle match_list \rangle \langle default_case \rangle \}$$
$$\begin{aligned}\langle match_list \rangle & ::= \langle matchee \rangle \\ & \quad | \langle match_list \rangle \langle matchee \rangle\end{aligned}$$
$$\begin{aligned}\langle matchee \rangle & ::= \langle match_case \rangle \\ & \quad | \langle regex_branch \rangle\end{aligned}$$
$$\langle match_case \rangle ::= 'case' \langle expression \rangle \langle case_condition \rangle ':' \langle block \rangle$$
$$\langle regex_branch \rangle ::= 'regex' \langle expression \rangle \langle as \rangle \langle case_condition \rangle ':' \langle block \rangle$$
$$\begin{aligned}\langle as \rangle & ::= 'as' \langle expression \rangle \\ & \quad | \epsilon\end{aligned}$$
$$\langle case_condition \rangle ::= '|' \langle expression \rangle \langle case_condition \rangle ::= \epsilon$$

5.4.7 Scan statement

$$\langle scan_statement \rangle ::= 'scan' '(' \langle expression \rangle ')' 'using' \{ \langle regex_list \rangle \langle default_case \rangle \}$$
$$\begin{aligned}\langle regex_list \rangle & ::= \langle regex_branch \rangle \\ & \quad | \langle regex_list \rangle \langle regex_branch \rangle\end{aligned}$$

5.4.8 Switch statement

A slightly different version of the normal switch-case statement. It auto breaks on hit and needs an expression instead of giving a matchee to switch on and then values in each case.

$$\langle switch_statement \rangle ::= 'switch' \{ \langle case_statements \rangle \langle default_case \rangle \}$$

$$\begin{aligned}
\langle case_statements \rangle & ::= \langle case_list \rangle \\
& \quad | \quad \epsilon \\
\langle case_list \rangle & ::= \langle case_statement \rangle \\
& \quad | \quad \langle case_list \rangle \langle case_statement \rangle \\
\langle case_statement \rangle & ::= \text{'case'} \langle expression \rangle \text{'.'} \langle block \rangle \\
\langle default_case \rangle & ::= \text{'default'} \text{'.'} \langle block \rangle \\
& \quad | \quad \epsilon
\end{aligned}$$

5.4.9 Try statement

$$\begin{aligned}
\langle try_statement \rangle & ::= \text{'try'} \text{'{' } \langle block \rangle \text{'}} \langle catches \rangle \\
\langle catches \rangle & ::= \langle catch_clause \rangle \\
& \quad | \quad \langle catches \rangle \langle catch_clause \rangle \\
\langle catch_clause \rangle & ::= \langle catch_type \rangle \text{'(' } \langle identifier \rangle \text{')' '{' } \langle block \rangle \text{'}} \\
\langle catch_type \rangle & ::= \text{'catch'} \\
& \quad | \quad \text{'catchUsr'} \\
& \quad | \quad \text{'catchLng'}
\end{aligned}$$

5.4.10 While-Loop statement

Typical while-loop.

$$\langle while_loop \rangle ::= \text{'while'} \text{'(' } \langle expression \rangle \text{')' '{' } \langle block \rangle \text{'}}$$

Part III

Setlx2py internals

6 Project structure

The structure of `setlx2py` follows the one of a typical Python module. In addition to that, `nose` is used for unit testing, which also demands a specific naming of the test folder and respective test files.

```
/setlx2py
├── setlx2py
│   ├── __init__.py
│   ├── setlx_ast.cfg
│   ├── setlx_ast.py
│   ├── setlx_transformer.py
│   ├── setlx_builtin.py
│   ├── setlx_codegen.py
│   ├── setlx_lexer.py
│   ├── setlx_parser.py
│   ├── setlx_semcheck.py
│   ├── setlx_util.py
│   └── builtin
│       ├── __init__.py
│       ├── setlx_functions.py
│       ├── setlx_list.py
│       ├── setlx_set.py
│       └── setlx_string.py
├── tests
│   ├── test_ast_transform.py
│   ├── test_builtin.py
│   ├── test_codegen.py
│   ├── test_execution.py
│   ├── test_lexer.py
│   ├── test_list.py
│   ├── test_parsable.py
│   ├── test_parser.py
│   └── test_set.py
```

Figure 5: Folders and files in `setlx2py`

6.1 Source code

When writing Python modules, the name of the folder where source files reside is the name of the module itself. That means, setlx2py files are imported with (as an example with setlx_util)

```
from setlx2py.setlx_util import *
```

6.1.1 setlx_cfg

The config file from which the AST Python code is generated. The process is explained in Section 8.

6.1.2 setlx_ast

This file contains the AST nodes used to represent a SetlX program internally. It is automatically generated from setlx_ast.cfg and should not be altered by hand.

6.1.3 setlx_transformer

This file defines the transformations which are made to the AST after it is generated by the parser. The process is explained in Section 10.

6.1.4 setlx_builtin

This folder holds all dependencies needed to actually run a file generated by setlx2py.

6.1.5 setlx_codegen

This file contains the code generator, which takes in an AST and outputs Python code. The process is explained in Section ??.

6.1.6 setlx_lexer

This file contains the lexer, which also contains the token definitions. It is used in the parser, which drives the lexing. The lexer is written with the PLY framework.

6.1.7 setlx_parser

This file contains the parser and the SetlX grammar. It reads in a SetlX program as a string and outputs an AST. It internally uses the lexer. The parser is written with the PLY framework.

6.1.8 setlx_semcheck

This file contains all functions to semantically check the generated sub-AST inside certain grammar rules in the parser. This is needed, since the grammar allows too much due to its LALR nature.

6.1.9 setlx_util

Central location to keep all functions which are called from at least to different files.

6.2 Test files

Nosetest, the test framework used, requires a folder named `tests`. Files prefixed with `test_` are -inter alia- test files. All files not recognized as test files are ignored when tests are run. The following paragraphs briefly describe the tests whose function is not directly obvious.

6.2.1 `test_builtin`

Contains tests for the builtin functions.

6.2.2 `test_execution`

Contains acceptance tests for the code generation. It is tested whether the execution of setlx2py-generated code yields the same result as the standard SetlX interpreter.

6.2.3 `test_list` & `test_set`

Tests for the `setlx_set` and `setlx_list`.

6.2.4 `test_parsable`

Contains acceptance tests for the parser. It is tested whether setlx2py can parse all the example SetlX code.

7 Testing

The software development method used to implement `setlx2py` was chosen to be test-driven development (TDD). The biggest reason for that was the fact that small changes in the parser might result in hard-to-find errors. In addition to that, the code generation has influence of many features which the SetlX language offers. A small change can render many of them wrong, and tests are used to catch them.

It proved to reduce the amount of debugging, and assures a high quality from start to end. Also, refactoring or extensions in the future (especially by new authors) which break the functionality are detected very early. Last but not least, the tests serve as documentation of the language.

7.1 Test-driven development

There are many definitions of test driven development, one of the most popular is the following by Robert Cecil Martin, a leader of the Agile and Clean Code Movement. It can be found in [1] and reads the following:

1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

Therefore, each token of the lexer, each grammar rule, each code generation, all builtins have corresponding tests, etc . . . which were written before actual functionality was implemented for them. `Setlx2py` has more lines of tests than implementation code.

It may be the case that different methodology would achieve the same result, but TDD worked well in this project.

7.2 Running the tests

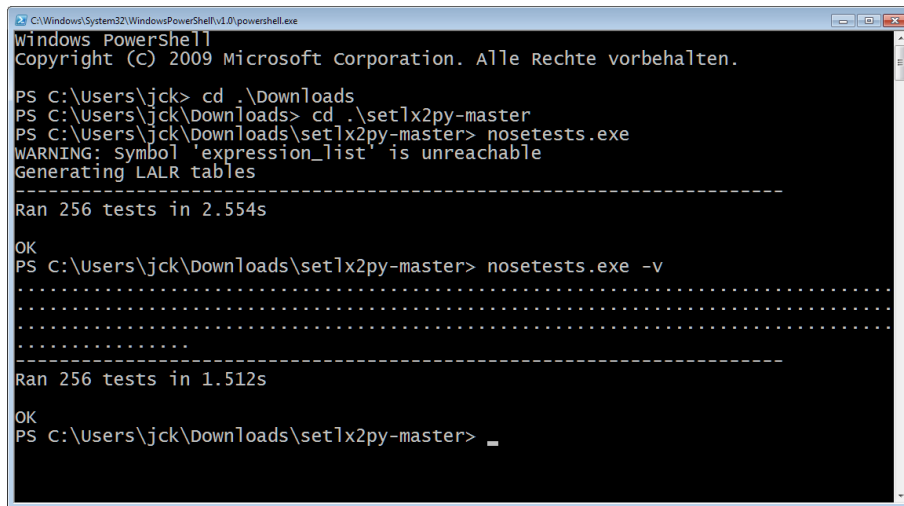
The tests are implemented using the nosetest framework for Python. They can be run with

nosetests

on Linux/Mac or

nosetests.exe

on Windows while being in the setlx2py root directory.



```
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. Alle Rechte vorbehalten.

PS C:\Users\jck> cd .\Downloads
PS C:\Users\jck\Downloads> cd .\setlx2py-master
PS C:\Users\jck\Downloads\setlx2py-master> nosetests.exe
WARNING: symbol 'expression_list' is unreachable
Generating LALR tables
-----
Ran 256 tests in 2.554s
OK
PS C:\Users\jck\Downloads\setlx2py-master> nosetests.exe -v
.....
-----
Ran 256 tests in 1.512s
OK
PS C:\Users\jck\Downloads\setlx2py-master> _
```

Figure 6: Output of nosetest: All tests passed

8 AST

The nodes for the AST used internally are generated from a config file to ease development. For the purpose of this project, the AST-generating part of another open-source compiler project⁵ was forked into another project⁶ for that. New functionality was added to ease testing them against expected trees. Further more, their vizualization was improved.

The idea behind ast-gen is to describe the nodes, their attributes and children in a config file. From it, a Python file containing one class for every node is generated and then can be used without any dependency to the ast-gen project. Therefore, it is only needed in development.

8.1 Configuration

Each entry in the config file is a sub-class `<name>` of `Node`, listing its attributes and child nodes. Each line contains the name of the class, followed by it attributes:

```
<Name>: [list, of, attributes]
```

The attributes can be of the following kind:

```
<name>*      - a child node
<name>**     - a sequence of child nodes
<name>       - an attribute.
```

Example:

```
BinaryOp: [op, left*, right*]
Constant: [type, value]
ExprList: [exprs**]
```

The file used in `setlx2py` can be found in `setlx2py/setlx_ast.cfg`.

8.2 Generation

After installing ast-gen, the Python file can be generated with the folling command in a Python shell:

```
from astgen.ast_gen import ASTCodeGenerator
gen = ASTCodeGenerator(**PATH_TO_CONFIG.cfg**)
with open(**PATH_TO_WHERE_TO_SAVE**, 'w') as f:
    gen.generate(f)
```

Alternatively, the Makefile delivered with `setlx2py` offers a nice shortcut:

```
make ast
```

⁵<https://github.com/eliben/pycparser>

⁶<https://github.com/Rentier/ast-gen>

9 Code generation

The code generator used is model based, as described in [2], p. 295ff. It is implemented with the visitor pattern: the AST is traversed depth-first, and one can specify actions for every AST node type.

Therefore, every AST node type used in setlx2py has a Python format string with filler (like printf in the C language), where the content is injected. Every node is translated on its own, and the resulting strings are simply concatenated in the `FileAST` rule. The name scheme for callbacks is `visit_ + <node name>`, for example `visit_Assignment`.

The following snippet is a very basic code generator. The visit method dynamically looks up the callback to use for the current node and executes them. In the assignment itself, the subtrees are constructed in place (for lhs and rhs), and insert the resulting strings into the template.

```
class Codegen(object):

    def visit(self, node):
        method = 'visit_' + node.__class__.__name__
        fun = getattr(self, method, self.generic_visit)(node)
        return fun

    def visit_FileAST(self, n):
        s = ''
        for stmt in n.stmts:
            s += self.visit(stmt) + '\n'
        return s

    def visit_Assignment(self, n):
        s = '{0} {1} {2}'
        op = n.op if n.op != ':' else '='
        lhs = self.visit(n.target)
        rhs = self.visit(n.right)

        s = '{0} {1} {2}'
        return s.format(lhs, op, rhs)
```

10 Transformation

In order to make code generation easier, the parser-generated AST is partially rewritten. Mostly, that is done because some constructs in SetlX and Python differ quite largely. This maybe could also be done in the parser, but the main reason for a dedicated transformation step is to decouple the components.

It was tried to avoid altering the parser in order to simplify code generation, as these components should be as simple and asindependent from each other as possible.

The implementation is similar to the one of code generation: the AST is traversed depth-first, and one can specify actions for every AST node type.

In this section, the two most interesting and complex cases are described.

10.1 Procedure definitions

In order to define a function in SetlX, one creates an anonymous function and binds it to a name via assignment.

```
signum := procedure(n) {  
    if(n > 0) {  
        return 1;  
    } else if(n < 0) {  
        return - 1;  
    } else {  
        return 0;  
    }  
};
```

In Python, when defining a function, it is automatically bound to that name in the current scope:

```
def signum(n):  
    if n > 0:  
        return 1  
    elif n < 0:  
        return - 1;  
    else:  
        return 0;
```

The AST for a procedure definition in setlx2py consists of the assignment node itself with name on the left- and the procedure on the right-hand side. As an example, the following procedure is discussed:

```
add := procedure(a,b) { return a + b; };
```

The corresponding AST is

```
('Assignment', ':=',  
 ('Identifier', 'add'),  
 ('Procedure', '', 'vanilla',  
  ('ParamList', ('Param', 'a'), ('Param', 'b'))),  
 ('Block',  
  ('Return',
```

```
(('BinaryOp', '+',
  ('Identifier', 'a'),
  ('Identifier', 'b')))))))
```

One can see that the name of the procedure, (which Python needs for declaring it) is to be found in the assignment node. In the code generation step, only the current node and its children are visible to a callback. That means, the procedure node would have no access to the procedure name. The simple solution is to rewrite the AST for this construct. The rule for that is therefore that if there is an assignment statement where the right-hand side is a procedure, the name is assigned to the procedure node and the assignment node is replaced with its right-hand side:

```
def visit_Assignment(self, n, p):
    if isinstance(stmt, Assignment) and isinstance(stmt.right, Procedure):
        assignment = stmt
        procedure = assignment.right
        procedure.name = assignment.target.name
        n.stmts[i] = procedure
```

The result of transformation is

```
(('Procedure', 'add', 'vanilla',
  ('ParamList', ('Param', 'a'), ('Param', 'b')),
  ('Block',
    ('Return',
      ('BinaryOp', '+',
        ('Identifier', 'a'),
        ('Identifier', 'b'))))))
```

10.2 Interpolation

String interpolation in SetlX offers an easy way to inline expressions into strings. It is especially a nice syntax to create strings from data.

```
s := "signum($n$) is $signum(n)$.";
```

As a string is enriched with expressions, these have to be parsed aswell. Therefore, interpolation is only available at compile time. Parsing of the inlined expressions is done after the AST has been generated by the parser. To be more specific, it is done in the transformation phase. The high-level view is very simple:

```
def visit_Interpolation(self, n, p):
    self.fill_interpolation(n)
    self.generic_visit(n, p)
```

The `fill_interpolation` extracts the expressions in the given string, parses them, and creates the string format with the expressions as arguments. The AST after parsing is

```
(('Assignment', ':=',
  ('Identifier', 's'),
  ('Interpolation',
```

Update
to
rewrit-
ten
visit_Assignme

```
(('Constant', 'literal', 'signum($n$) is $signum(n)$.'),
 ('ExprList',)))
```

The result of transformation is

```
(('Assignment', ':=',
 ('Identifier', 's'),
 ('Interpolation',
 ('Constant', 'literal', 'signum({0}) is {1}.'),
 ('ExprList',
 ('Identifier', 'n'),
 ('Call',
 ('Identifier', 'signum'),
 ('ArgumentList',
 ('Identifier', 'n'))))))))
```

Finally, the generated Python code is

```
s = "signum({0}) is {1}.".format(n, signum(n))
```

11 Known issues

There is still work to be done:

- Nested collections in iterators
- Scan
- Try/Catch
- Backtrack
- Many of the builtins

References

- [1] **Robert Martin**. *The Three Rules Of Tdd*. <http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd>. Oct. 2005.
- [2] **Terence Parr**. *Language Implementation Patterns: Create Your Own Domain-Specific and General Programming Languages (Pragmatic Programmers)*. Pragmatic Bookshelf, 2010. ISBN: 193435645X.
- [3] **Python Software Foundation**. *The Python Language Reference*. <https://docs.python.org/2/reference/>. May 2014.