DHBW Mannheim

Student research project

TINF11AI-BC

Implementing a SetlX to Pyton compiler

Author:
Jan-Christoph Klie

Supervisor:
Prof. Dr. Karl Stroetmann

May 19, 2014

Contents

List of Tables

List of Figures

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.

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To deal with a target platform which is not described here, our this installation guide becomes outdated, the dependencies necessary to run setlx2py are now explained.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

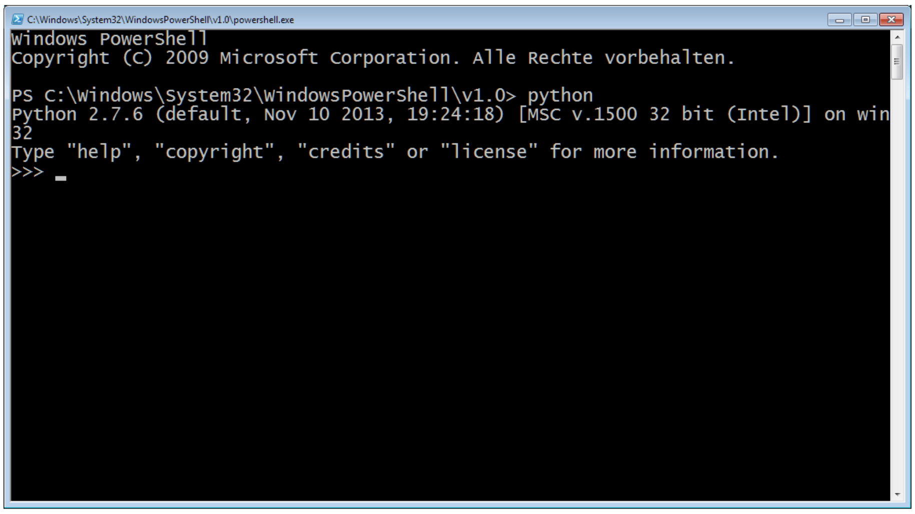
in the shell.

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.github.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



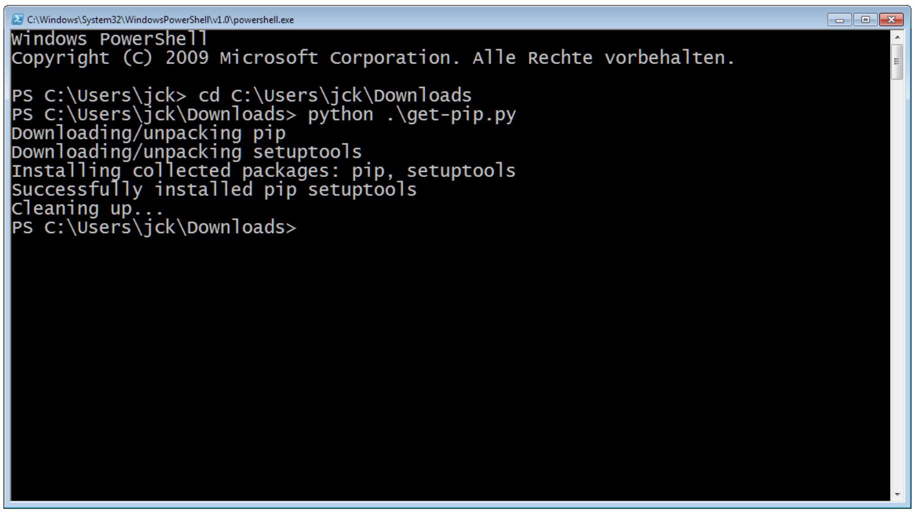
```
C:\Windows\System32\WindowsPowerShell\v1.0\powershell.exe
Windows PowerShell
Copyright (C) 2009 Microsoft Corporation. Alle Rechte vorbehalten.

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 people offer them 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 dependencies

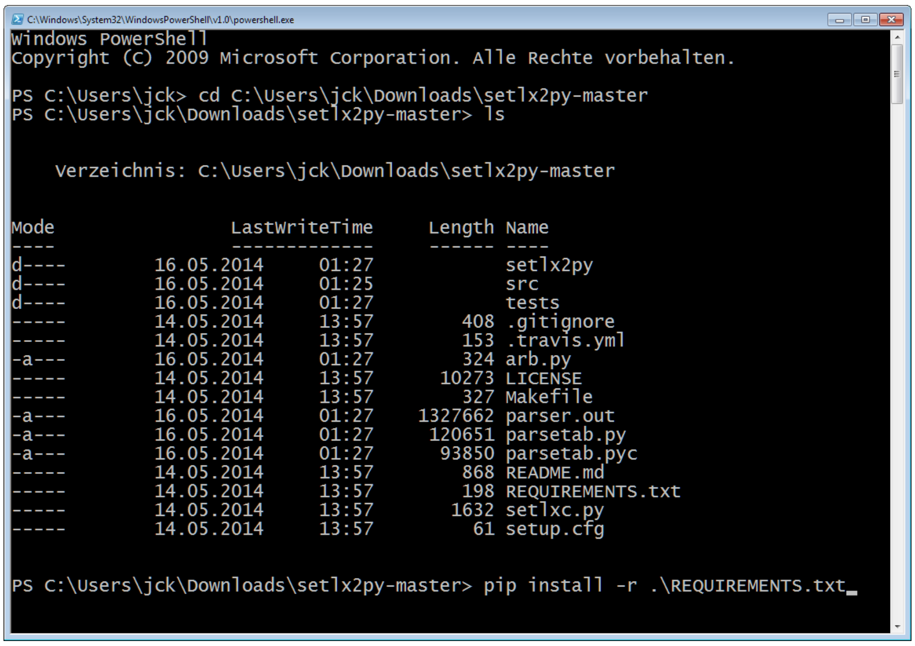
Download the setlx2py archive from

`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.



```
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

2.5 Use setlx2py

Now that all prequisites are installed, the actual compiler can be downloaded. This step uses the previously installed package manager, so it only requires one command to install setlx2py. Grab an open shell, and issue the following command:

```
# pip install git+git://github.com/Rentier/setlx2py.git@master
```

3 Linux

4 Mac OS

Part II

The SetlX language

5 Lexical analysis

5.1 Line structure

A SetlX program is built of logical lines.

5.1.1 Logical lines

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

5.1.2 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.

5.1.3 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 `*/`.

5.1.4 Blank lines

Lines that only contain whitespace are ignored.

5.1.5 Whitespace

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

5.2 Identifiers and keywords

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

5.2.1 Identifier

Identifiers are described by the following lexical definitions:

⟨*identifier*⟩	::=	⟨*lowercase*⟩ ⟨*seq*⟩ | '_'
⟨*seq*⟩	::=	⟨*seq*⟩ ⟨*char*⟩
	|	ϵ
⟨*char*⟩	::=	⟨*lowercase*⟩ | ⟨*uppercase*⟩ | ⟨*digit*⟩ | '_'
⟨*lowercase*⟩	::=	'a'...'z'
⟨*uppercase*⟩	::=	'A'...'Z'
⟨*digit*⟩	::=	'0'...'9'

5.2.2 Functor

Functors are described by the following lexical definitions:

⟨*functor*⟩	::=	⟨*uppercase*⟩ ⟨*seq*⟩

5.3 Keywords

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

Update key-words

`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`		

5.4 Literals

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

5.4.1 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.

⟨*interpolation*⟩ ::= ‘`$`’ ⟨*expression*⟩ ‘`$`’

Escaping

SetlX supports the same escape sequences as the language C.

5.4.2 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

⟨*integer*⟩ ::= ⟨*nonzerodigit*⟩ ⟨*digits-or-empty*⟩ | ‘`0`’

⟨*nonzerodigit*⟩ ::= ‘`1`’...‘`9`’

⟨*digits*⟩ ::= ⟨*digits*⟩ ⟨*digit*⟩
| ⟨*digit*⟩

⟨*digits-or-empty*⟩ ::= ⟨*digits*⟩ | ϵ

Floating point literals

Floating point literals are described by the following lexical definitions:

⟨*floatnumber*⟩ ::= ⟨*pointfloat*⟩ | ⟨*exponentfloat*⟩

⟨*pointfloat*⟩ ::= ⟨*fraction*⟩
| ⟨*intpart*⟩ ⟨*fraction*⟩
| ⟨*intpart*⟩ ‘`.`’

⟨*exponentfloat*⟩ ::= ⟨*significand*⟩ ⟨*exponent*⟩

⟨*significand*⟩ ::= ⟨*intpart*⟩ | ⟨*pointfloat*⟩

⟨*intpart*⟩ ::= ⟨*digits*⟩

⟨*fraction*⟩ ::= ‘`.`’ ⟨*digits*⟩

⟨*exponent*⟩ ::= ⟨*e*⟩ ⟨*sign*⟩ ⟨*digits*⟩

⟨*e*⟩ ::= ‘`e`’ | ‘`E`’

⟨*sign*⟩ ::= ‘`+`’ | ‘`-`’ | ϵ

5.5 Operators

The following tokens are operators:

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

5.6 Delimiters

The following tokens serve as delimiters in the grammar:

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

6 Execution model

6.1 Scoping

7 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 model used is heavily borrowed from the Python 2.7 grammar. It can be found in

Pyton Grammar

7.1 Top-level components

All input read from files has the same form:

⟨*S*⟩	::=	⟨*file_input*⟩
⟨*file_input*⟩	::=	⟨*statement_list*⟩
	|	⟨*expression*⟩
⟨*statement_list*⟩	::=	⟨*statement*⟩
	|	⟨*statement_list*⟩ ⟨*statement*⟩
⟨*statement*⟩	::=	⟨*simple_statement*⟩ ‘**;**’
	|	⟨*compound_statement*⟩
⟨*expression_list*⟩	::=	⟨*expression*⟩
	|	⟨*expression_list*⟩ ‘**,**’ ⟨*expression*⟩
⟨*block*⟩	::=	⟨*statement_list*⟩
	|	ϵ

7.2 Expressions

This section explains the elements occuring in SetlX expressions.

```
⟨expression⟩          ::= ⟨implication⟩
                        | ⟨lambda_definition⟩
                        | ⟨implication⟩ '<==>' ⟨implication⟩
                        | ⟨implication⟩ '<!=>' ⟨implication⟩

⟨implication⟩         ::= ⟨disjunction⟩
                        | ⟨disjunction⟩ '=>' ⟨disjunction⟩

⟨disjunction⟩         ::= ⟨conjunction⟩
                        | ⟨disjunction⟩ '||' ⟨conjunction⟩

⟨conjunction⟩         ::= ⟨comparison⟩
                        | ⟨conjunction⟩ '&&' ⟨comparison⟩

⟨comparison⟩          ::= ⟨sum⟩
                        | ⟨sum⟩ '==' ⟨sum⟩
                        | ⟨sum⟩ '!=' ⟨sum⟩
                        | ⟨sum⟩ '<' ⟨sum⟩
                        | ⟨sum⟩ '<=' ⟨sum⟩
                        | ⟨sum⟩ '>' ⟨sum⟩
                        | ⟨sum⟩ '>=' ⟨sum⟩
                        | ⟨sum⟩ 'in' ⟨sum⟩
                        | ⟨sum⟩ 'notin' ⟨sum⟩

⟨sum⟩                 ::= ⟨product⟩
                        | ⟨sum⟩ '+' ⟨product⟩
                        | ⟨sum⟩ '-' ⟨product⟩

⟨product⟩             ::= ⟨reduce⟩
                        | ⟨product⟩ '*' ⟨reduce⟩
                        | ⟨product⟩ '/' ⟨reduce⟩
                        | ⟨product⟩ '\' ⟨reduce⟩
                        | ⟨product⟩ '%' ⟨reduce⟩
                        | ⟨product⟩ '<>' ⟨reduce⟩

⟨reduce⟩              ::= ⟨unary_expression⟩
                        | '\+' ⟨unary_expression⟩
                        | '\*' ⟨unary_expression⟩

⟨unary_expression⟩    ::= ⟨power⟩
                        | '\+' ⟨unary_expression⟩
                        | '\*' ⟨unary_expression⟩
                        | '#' ⟨unary_expression⟩
                        | '-' ⟨unary_expression⟩
                        | '@' ⟨unary_expression⟩
```

```
                      |   '!' ⟨unary_expression⟩
                      |   ⟨quantor⟩
                      |   ⟨term⟩

⟨power⟩              ::=  ⟨primary⟩
                      |   ⟨primary⟩ '**' ⟨power⟩

⟨primary⟩            ::=  ⟨atom⟩
                      |   ⟨attributeref⟩
                      |   ⟨subscription⟩
                      |   ⟨slicing⟩
                      |   ⟨procedure⟩
                      |   ⟨call⟩
                      |   ⟨primary⟩ '!'
```

lambda

```
⟨lambda_definition⟩  ::=  ⟨lambda_parameters⟩ '|->' ⟨expression⟩

⟨lambda_parameters⟩  ::=  ⟨identifier⟩
                      |   ⟨list_display⟩
```

quantor

```
⟨quantor⟩            ::=  'forall' '(' ⟨iterator_chain⟩ '|' ⟨expression⟩ ')'
                      |   'exists' '(' ⟨iterator_chain⟩ '|' ⟨expression⟩ ')'
```

term

```
⟨term⟩               ::=  'TERM' '(' ⟨argument_list⟩ ')'
                      |   'TERM' '(' ')'
```

7.2.1 Atom

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

```
⟨atom⟩               ::=  ⟨identifier⟩
                      |   ⟨literal⟩
                      |   ⟨enclosure⟩

⟨identifier⟩         ::=  'IDENTIFIER'
                      |   '_'
```

Literals

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

```
⟨literal⟩            ::=  ⟨stringliteral⟩
                      |   ⟨integer⟩
```

	|	⟨*floatnumber*⟩
	|	⟨*boolean*⟩

⟨*stringliteral*⟩	::=	`'STRING'`
	|	`'LITERAL'`
	|	`'INTERPOLATION'`

⟨*integer*⟩ ::= `'INTEGER'`

⟨*floatnumber*⟩ ::= `'DOUBLE'`

⟨*boolean*⟩	::=	`'true'`
	|	`'false'`

7.2.2 Primaries

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

Attributeref

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

⟨*attributeref*⟩ ::= ⟨*primary*⟩ `'.'` ⟨*identifier*⟩

Subscription

A subscription retrieves an item of an indexable:

⟨*subscription*⟩ ::= ⟨*primary*⟩ `'['` ⟨*expression*⟩ `']'`

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.

⟨*slicing*⟩ ::= ⟨*primary*⟩ `'['` ⟨*lower_bound*⟩ `'..'` ⟨*upper_bound*⟩ `']'`

⟨*lower_bound*⟩	::=	⟨*expression*⟩
	|	ϵ

⟨*upper_bound*⟩	::=	⟨*expression*⟩
	|	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.

⟨*procedure*⟩ ::= ‘`procedure`’ ‘`(`’ parameter_list ‘`)`’ ‘`{`’ ‘`block`’ ‘`}`’
| ‘`cProcedure`’ ‘`(`’ parameter_list ‘`)`’ ‘`{`’ ‘`block`’ ‘`}`’

⟨*parameter_list*⟩ ::= ⟨*params*⟩
| ϵ

⟨*params*⟩ ::= ⟨*procedure_param*⟩
| ⟨*params*⟩ ‘`,`’ ⟨*procedure_param*⟩

⟨*procedure_param*⟩ ::= ⟨*identifier*⟩

Call

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

⟨*call*⟩ ::= ⟨*primary*⟩ ‘`(`’ ⟨*argument_list*⟩ ‘`)`’
| ⟨*primary*⟩ ‘`(`’ ‘`)`’

⟨*argument_list*⟩ ::= ⟨*expression*⟩
| ⟨*argument_list*⟩ ‘`,`’ ⟨*expression*⟩

7.2.3 Enclosures

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

⟨*enclosure*⟩ ::= ⟨*parenth_form*⟩
| ⟨*set_range*⟩
| ⟨*list_range*⟩
| ⟨*set_display*⟩
| ⟨*list_display*⟩
| ⟨*set_comprehension*⟩
| ⟨*list_comprehension*⟩

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.

⟨*parenth_form*⟩ ::= ‘`(`’ ⟨*expression*⟩ ‘`)`’

Comprehensions

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

⟨*set_comprehension*⟩ ::= ‘`{`’ ⟨*expression*⟩ ‘`:`’ ⟨*iterator_chain*⟩ ⟨*comprehension_condition*⟩ ‘`}`’

⟨*list_comprehension*⟩ ::= ‘`[`’ ⟨*expression*⟩ ‘`:`’ ⟨*iterator_chain*⟩ ⟨*comprehension_condition*⟩ ‘`]`’

```
⟨comprehension_condition⟩ ::= ‘|’ ⟨expression⟩
                            |  ϵ
```

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.

```
⟨set_range⟩       ::= ‘{’ ⟨expression⟩ ‘..’ ⟨expression⟩ ‘}’
                    |  ‘{’ ⟨expression⟩ ‘,’ ⟨expression⟩ ‘..’ ⟨expression⟩ ‘}’

⟨list_range⟩      ::= ‘[’ ⟨expression⟩ ‘..’ ⟨expression⟩ ‘]’
                    |  ‘[’ ⟨expression⟩ ‘,’ ⟨expression⟩ ‘..’ ⟨expression⟩ ‘]’
```

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.

```
⟨set_display⟩     ::= ‘{’ ⟨expression⟩ ‘}’
                    |  ‘{’ ⟨expression⟩ ‘,’ argument_list ‘}’
                    |  ‘{’ ‘}’
                    |  ‘{’ ⟨expression⟩ ‘|’ ⟨expression⟩ ‘}’
                    |  ‘{’ ⟨expression⟩ ‘,’ argument_list ‘|’ ⟨expression⟩ ‘}’

⟨list_display⟩    ::= ‘[’ ⟨expression⟩ ‘]’
                    |  ‘[’ ⟨expression⟩ ‘,’ argument_list ‘]’
                    |  ‘[’ ‘]’
                    |  ‘[’ ⟨expression⟩ ‘|’ ⟨expression⟩ ‘]’
                    |  ‘[’ ⟨expression⟩ ‘,’ argument_list ‘|’ ⟨expression⟩ ‘]’
```

7.3 Simple statements

Simple statements form a single logical line and therefore have to end with a semicolon.

⟨*simple_statement*⟩ ::= ⟨*assert_statement*⟩
| ⟨*assignment_statement*⟩
| ⟨*augmented_assign_statement*⟩
| ⟨*backtrack_statement*⟩
| ⟨*break_statement*⟩
| ⟨*continue_statement*⟩
| ⟨*exit_statement*⟩
| ⟨*expression_statement*⟩
| ⟨*return_statement*⟩

7.3.1 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.

⟨*assert_statement*⟩ ::= '`assert`' '`(`' ⟨*expression*⟩ '`COMMA`' ⟨*expression*⟩ '`)`'

7.3.2 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.

⟨*assignment_statement*⟩ ::= ⟨*target*⟩ '`:=`' ⟨*expression*⟩

⟨*target*⟩ ::= ⟨*expression*⟩

7.3.3 Augmented assignment statement

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

⟨*augmented_assign_statement*⟩ ::= ⟨*augtarget*⟩ ⟨*augop*⟩ ⟨*expression*⟩

⟨*augtarget*⟩ ::= ⟨*identifier*⟩
| ⟨*attributeref*⟩
| ⟨*subscription*⟩

⟨*augop*⟩ ::= '`+=`' | '`-=`' | '`*=`' | '`/=`' | '`\=`' | '`%=`'

7.3.4 Backtrack statement

⟨*backtrack_statement*⟩ ::= '`backtrack`'

7.3.5 Break statement

$\langle$*break_statement*$\rangle$ ::= '`break`'

7.3.6 Continue statement

$\langle$*continue_statement*$\rangle$ ::= '`continue`'

7.3.7 Exit statement

$\langle$*exit_statement*$\rangle$ ::= '`exit`'

7.3.8 Expression statement

An expression statement evaluates the given expression.

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

7.3.9 Return statement

$\langle$*return_statement*$\rangle$ ::= '`return`'
| '`return`' $\langle$*expression*$\rangle$

7.4 Compound statements

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

```
⟨compound_statement⟩ ::= ⟨check_statement⟩
                      |  ⟨class⟩
                      |  ⟨do_while_loop⟩
                      |  ⟨for_loop⟩
                      |  ⟨if_statement⟩
                      |  ⟨match_statement⟩
                      |  ⟨scan_statement⟩
                      |  ⟨switch_statement⟩
                      |  ⟨try_statement⟩
                      |  ⟨while_loop⟩
```

7.4.1 Check statement

Belongs to backtracking.

```
⟨check_statement⟩     ::= ‘check’ ‘{’ ⟨block⟩ ‘}’
```

7.4.2 Class statement

```
⟨class⟩               ::= ‘class’ ⟨identifier⟩ ‘(’ ⟨parameter_list⟩ ‘)’ ‘{’ ⟨block⟩ ⟨static_block⟩
                          ‘}’

⟨static_block⟩        ::= ‘static’ ‘{’ ⟨block⟩ ‘}’
                      |  ϵ
```

7.4.3 Do-While statement

Typical do-while loop.

```
⟨do_while_loop⟩       ::= ‘do’ ‘{’ ⟨block⟩ ‘}’ ‘while’ ‘(’ ⟨expression⟩ ‘)’ ‘;’
```

7.4.4 For-Loop statement

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

```
⟨for_loop⟩            ::= ‘for’ ‘(’ ⟨iterator_chain⟩ ‘)’ ‘{’ ⟨block⟩ ‘}’

⟨iterator⟩            ::= ⟨comparison⟩

⟨iterator_chain⟩      ::= ⟨iterator⟩
                      |  ⟨iterator_chain⟩ ‘,’ ⟨iterator⟩
```

7.4.5 If statement

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

⟨*if_statement*⟩ ::= ‘`if`’ ‘`(`’ ⟨*expression*⟩ ‘`)`’ ‘`{`’ ⟨*block*⟩ ‘`}`’
| ‘`if`’ ‘`(`’ ⟨*expression*⟩ ‘`)`’ ‘`{`’ ⟨*block*⟩ ‘`}`’ ‘`else`’ ‘`{`’ ⟨*block*⟩ ‘`}`’
| ‘`if`’ ‘`(`’ ⟨*expression*⟩ ‘`)`’ ‘`{`’ ⟨*block*⟩ ‘`}`’ ‘`else`’ ⟨*if_statement*⟩

7.4.6 Match statement

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

⟨*match_statement*⟩ ::= ‘`match`’ ‘`(`’ ⟨*expression*⟩ ‘`)`’ ‘`{`’ ⟨*match_list*⟩ ⟨*default_case*⟩ ‘`}`’

⟨*match_list*⟩ ::= ⟨*matchee*⟩
| ⟨*match_list*⟩ ⟨*matchee*⟩

⟨*matchee*⟩ ::= ⟨*match_case*⟩
| ⟨*regex_branch*⟩

⟨*match_case*⟩ ::= ‘`case`’ ⟨*expression*⟩ ⟨*case_condition*⟩ ‘`:`’ ⟨*block*⟩

⟨*regex_branch*⟩ ::= ‘`regex`’ ⟨*expression*⟩ ⟨*as*⟩ ⟨*case_condition*⟩ ‘`:`’ ⟨*block*⟩

⟨*as*⟩ ::= ‘`as`’ ⟨*expression*⟩
| ϵ

⟨*case_condition*⟩ ::= ‘`|`’ ⟨*expression*⟩ ⟨*case_condition*⟩ ::= ϵ

7.4.7 Scan statement

⟨*scan_statement*⟩ ::= ‘`scan`’ ‘`(`’ ⟨*expression*⟩ ‘`)`’ ‘`using`’ ‘`{`’ ⟨*regex_list*⟩ ⟨*default_case*⟩ ‘`}`’

⟨*regex_list*⟩ ::= ⟨*regex_branch*⟩
| ⟨*regex_list*⟩ ⟨*regex_branch*⟩

7.4.8 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.

⟨*switch_statement*⟩ ::= ‘`switch`’ ‘`{`’ ⟨*case_statements*⟩ ⟨*default_case*⟩ ‘`}`’

⟨*case_statements*⟩	::=	⟨*case_list*⟩
	|	epsilon
⟨*case_list*⟩	::=	⟨*case_statement*⟩
	|	⟨*case_list*⟩ ⟨*case_statement*⟩
⟨*case_statement*⟩	::=	‘`case`’ ⟨*expression*⟩ ‘`:`’ ⟨*block*⟩
⟨*default_case*⟩	::=	‘`default`’ ‘`:`’ ⟨*block*⟩
	|	ϵ

7.4.9 Try statement

⟨*try_statement*⟩	::=	‘`try`’ ‘`{`’ ⟨*block*⟩ ‘`}`’ ⟨*catches*⟩
⟨*catches*⟩	::=	⟨*catch_clause*⟩
	|	⟨*catches*⟩ ⟨*catch_clause*⟩
⟨*catch_clause*⟩	::=	⟨*catch_type*⟩ ‘`(`’ ⟨*identifier*⟩ ‘`)`’ ‘`{`’ ⟨*block*⟩ ‘`}`’
⟨*catch_type*⟩	::=	‘`catch`’
	|	‘`catchUsr`’
	|	‘`catchLng`’

7.4.10 While-Loop statement

Typical while-loop.

⟨*while_loop*⟩	::=	‘`while`’ ‘`(`’ ⟨*expression*⟩ ‘`)`’ ‘`{`’ ⟨*block*⟩ ‘`}`’

Part III

Setlx2py internals

8 Project structure

```
/setlx2py
└── setlx2py
    ├── __init__.py
    ├── 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
```

9 Testing

10 Running the tests

11 AST

12 Transformations

13 Code generation

14 Execution

15 Known issues