DHBW Mannheim

Student research project

TINF11AI-BC

Implementing a SetlX to Python compiler

Author:
Jan-Christoph Klie

Supervisor:
Prof. Dr. Karl Stroetmann

July 25, 2014

Contents

List of Figures

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To deal with a target platform which is not described here, or this installation guide becomes outdated, the dependencies necessary to run setlx2py are now explained.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

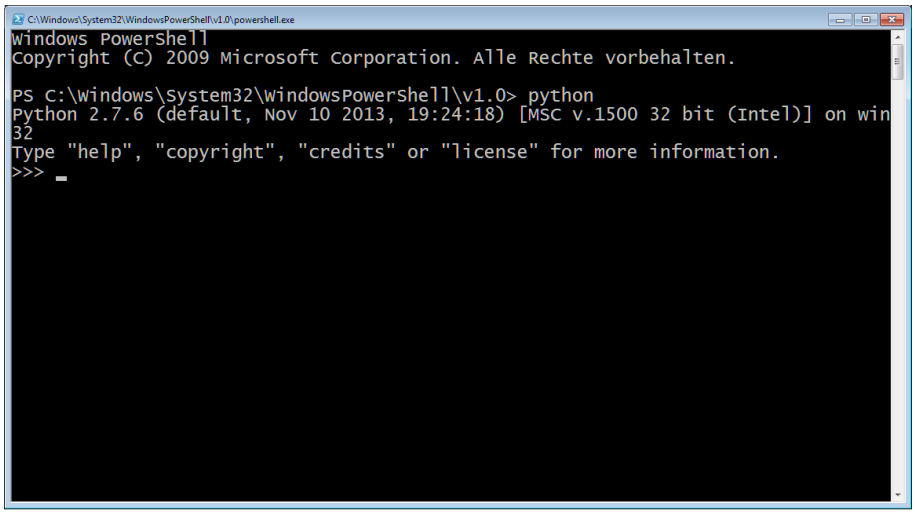

Figure 1: Running python from a shell without fully specified path

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

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

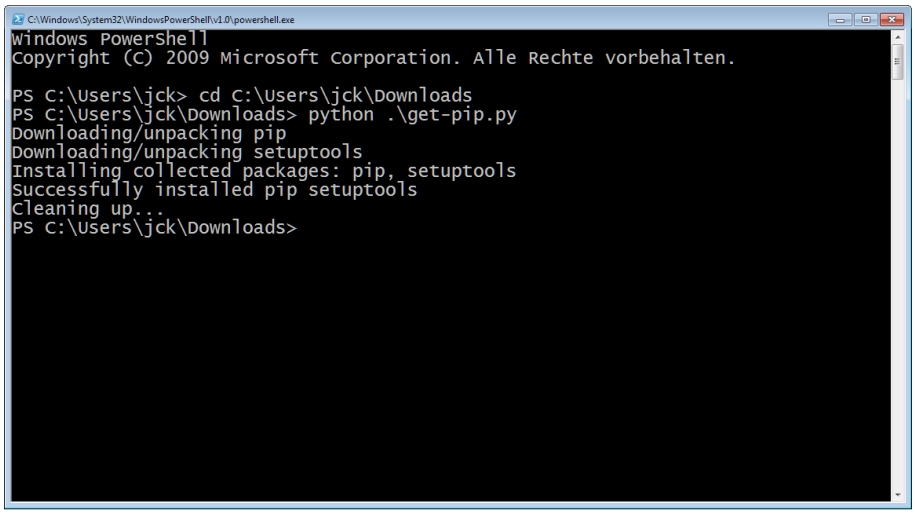

Figure 2: Installing the package manager pip

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.

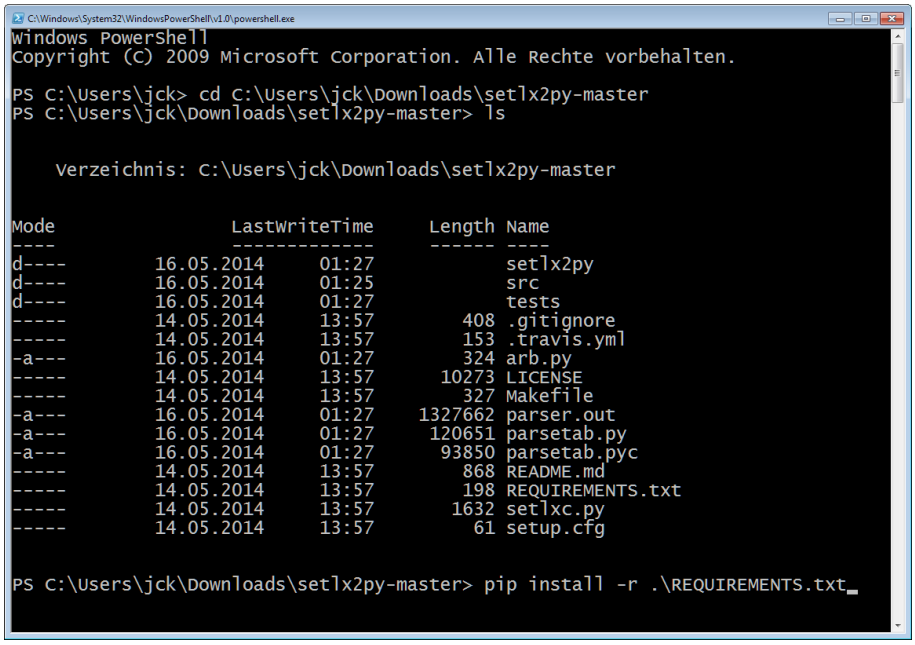

Figure 3: Installing the dependencies of setlx2py with pip

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.

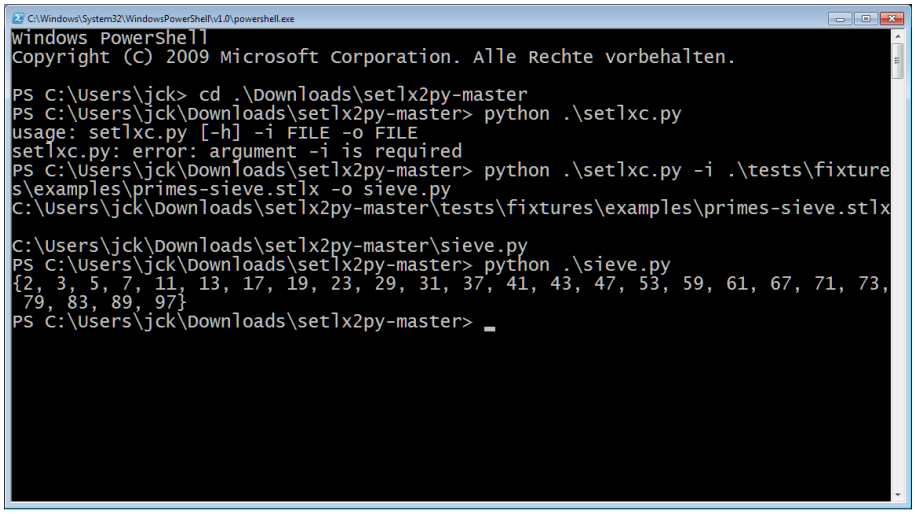

Figure 4: Using setlx2py to compile SetlX and running the generated Python

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:

⟨*identifier*⟩ ::= ⟨*lowercase*⟩ ⟨*seq*⟩ | '_'

⟨*seq*⟩ ::= ⟨*seq*⟩ ⟨*char*⟩
| ϵ

⟨*char*⟩ ::= ⟨*lowercase*⟩ | ⟨*uppercase*⟩ | ⟨*digit*⟩ | '_'

⟨*lowercase*⟩ ::= 'a'...'z'

⟨*uppercase*⟩ ::= 'A'...'Z'

⟨*digit*⟩ ::= '0'...'9'

4.2.2 Functor

Functors are described by the following lexical definitions:

⟨*functor*⟩ ::= ⟨*uppercase*⟩ ⟨*seq*⟩

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.

⟨*interpolation*⟩ ::= '$' ⟨*expression*⟩ '$'

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

⟨*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

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 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' '(' ')'

5.2.1 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'

5.2.2 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*⟩

5.2.3 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*⟩ '`]`'

5.3 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*⟩

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.

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

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.

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

⟨*target*⟩ ::= ⟨*expression*⟩

5.3.3 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*⟩ ::= '`+=`' | '`-=`' | '`*=`' | '`/=`' | '`\=`' | '`%=`'

5.3.4 Backtrack statement

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

5.3.5 Break statement

⟨*break_statement*⟩ ::= '`break`'

5.3.6 Continue statement

⟨*continue_statement*⟩ ::= '`continue`'

5.3.7 Exit statement

⟨*exit_statement*⟩ ::= '`exit`'

5.3.8 Expression statement

An expression statement evaluates the given expression.

⟨*expression_statement*⟩ ::= ⟨*expression*⟩

5.3.9 Return statement

⟨*return_statement*⟩ ::= '`return`'
| '`return`' ⟨*expression*⟩

5.4 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*⟩

5.4.1 Check statement

Belongs to backtracking.

⟨*check_statement*⟩ ::= '`check`' '`{`' ⟨*block*⟩ '`}`'

5.4.2 Class statement

⟨*class*⟩ ::= '`class`' ⟨*identifier*⟩ '`(`' ⟨*parameter_list*⟩ '`)`' '`{`' ⟨*block*⟩ ⟨*static_block*⟩ '`}`'

⟨*static_block*⟩ ::= '`static`' '`{`' ⟨*block*⟩ '`}`'
| ϵ

5.4.3 Do-While statement

Typical do-while loop.

⟨*do_while_loop*⟩ ::= '`do`' '`{`' ⟨*block*⟩ '`}`' '`while`' '`(`' ⟨*expression*⟩ '`)`' '`;`'

5.4.4 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*⟩

5.4.5 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*⟩

5.4.6 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

5.4.7 Scan statement

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

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

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.

⟨*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

5.4.9 Try statement

⟨*try_statement*⟩ ::= '`try`' '`{`' ⟨*block*⟩ '`}`' ⟨*catches*⟩

⟨*catches*⟩ ::= ⟨*catch_clause*⟩
| ⟨*catches*⟩ ⟨*catch_clause*⟩

⟨*catch_clause*⟩ ::= ⟨*catch_type*⟩ '`(`' ⟨*identifier*⟩ '`)`' '`{`' ⟨*block*⟩ '`}`'

⟨*catch_type*⟩ ::= '`catch`'
| '`catchUsr`'
| '`catchLng`'

5.4.10 While-Loop statement

Typical while-loop.

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

Part III

Setlx2py internals

6 Project structure

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

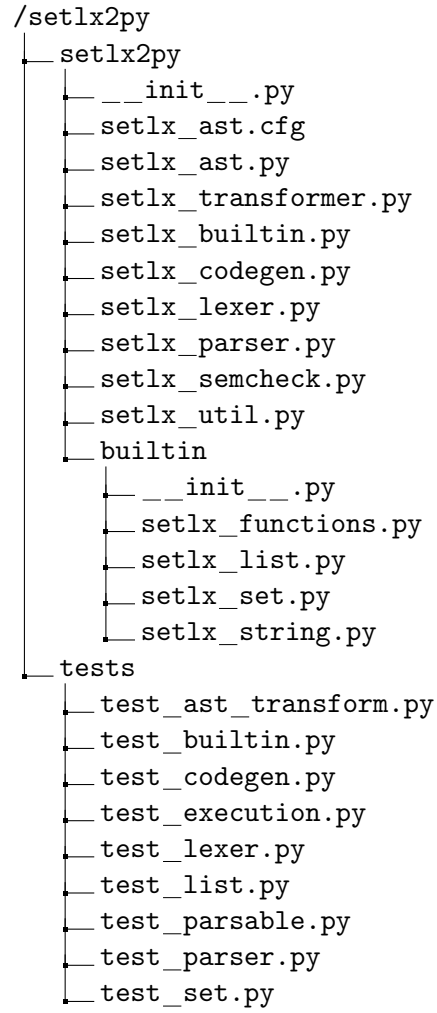

Figure 5: Folders and files in setlx2py

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 functionallity 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 fucntionality 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.

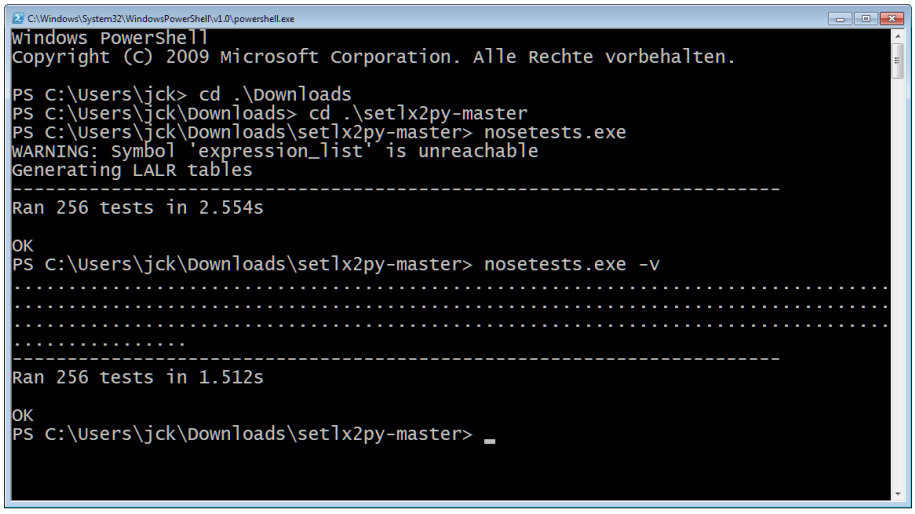

Figure 6: Output of nosetest: All tests passed

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 project[5] was forked into another project [6] for that. New functionallity 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 dynamicalls 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 an 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 is 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',
      ('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.