IST FP6-004779

PYPY

Researching a Highly Flexible and Modular Language Platform and Implementing it by Leveraging the Open Source Python Lanugage and Community

STREP

IST Priority 2

D04.3: Report about the parser and bytecode compiler implementation

Due date of deliverable: September 2005

Actual Submission date: 23th December 2005

Start date of Project: 1st December 2005   **Duration: 2 years**

Lead Contractor of this WP: Strakt

Authors: Adrien Di Mascio, Ludovic Aubry (Logilab)

Revision: final

Dissemination Level: PU (Public)

PyPy D04.3: Parser and Bytecode Compiler

Abstract

This document contains a report about the implementation of the PyPy parser and bytecode compiler implementations.

Contents

1 Executive Summary

The PyPy parser and bytecode compiler generates an Abstract Syntax Tree (AST) from the Python source code that is fed to it. Once the AST has been generated, the compiler traverses the tree to produce bytecode.

2 Introduction

2.1 Purpose of this Document

This document describes the implementation of the parser and compiler included with PyPy. It details the goals of the implementation and the design decisions made to achieve these goals.

2.2 Scope of this Document

This document is a design document targeted at the parser and compiler part. Other components of PyPy are described briefly only if interacting with the compiler and parser.

2.3 Related Documents

This document has been prepared in collaboration with other tasks in work packages 4 and 5. Studying the following deliverables is recommended before reading this document:

- D4.2 Complete Python Implementation on top of CPython

3 Goals

Early version of PyPy used the CPython compiler to compile Python source code into byte-code. The main focus was to have a working bytecode interpreter. Since one of the objectives of the project is to have a complete Python interpreter written in Python, it was necessary to write the source code to bytecode compiler in Python too.

A second objective is that the PyPy interpreter is to be translatable into C code, which means that all of the interpreter needs to be written in a subset of the Python language known as RPython (for Restricted Python).

Some work packages of phase 2 have a need for a means to easily extend Python syntax in order to allow easier experimentation with potential additionnal features. For this reason we wanted the design of the compiler/parser to be as flexible as possible to allow customization of the Python grammar itself as well as customization of the bytecode translation.

The needs here are twofold:

- extending the grammar, so that more language features are available, but still using existing bytecode

- eventually extend the bytecode generation

To summarize the design presented obeys the following requirements:

- flexible
- translatable
- extensible

4   Language Parsing Primer

This section introduces the reader to the terms and concepts used throughout this document.

4.1   BNF Grammars

BNF or Backus-Naur form (http://en.wikipedia.org/wiki/Backus-Naur_form) is a standard way of writing context free grammars. It is a meta-language in a sense since it is a language used to describe languages.

Several variants of the syntax exist and for the purpose of this document we will present the syntax used to describe Python's language which is Extended BNF, that is a BNF syntax with extra notations to represent repetitions of rules.

4.1.1   Python's Description of a Grammar

As a word of introduction we might explain what a grammar is: it is a set of rules which describe precisely the rules that determine if a sentence (or program) is syntactically correct.

A grammar rule looks like (taken from Python's grammar):

```
eval_input: testlist NEWLINE* ENDMARKER
```

This rule says that the symbol `eval_input` is composed of a sequence of three symbols: `testlist`, `NEWLINE` and `ENDMARKER`. This is not entirely correct since the star * has a special meaning saying `NEWLINE` can appear zero or more (any number of) times in the sequence. so any of these *productions* are correct:

```
testlist ENDMARKER
testlist NEWLINE ENDMARKER
testlist NEWLINE  NEWLINE  ENDMARKER
```

and so on...

Now we can also remark that `testlist` is lowercased while NEWLINE and ENDMARKER are both uppercased. This is a convention used to mark the difference between *terminal* and *non-terminal* symbols. A terminal symbol is a symbol that directly represents a token in the source program. In this example NEWLINE represents the character `\n` that is the ASCII representation of a carriage return.

`testlist` being lower case means that there is another rule describing the *production* of the symbol `testlist` and in fact, looking further into the grammar, we find:

```
testlist: test (',' test)* [',']
```

Note: This notation is specific to Python's grammar but you can find the same concepts anywhere.

Here we can see other symbols: `[`, `]`, `(`, `)` and `','`:

- The parentheses are used to group symbols so that the `*` applies to the group.

- The brackets have a meaning similar to the star: everything enclosed in the brackets is optional. In other words, the symbols in the brackets can appear zero or one (and only one) time.

- The quotes are just a shortcut to describe a token by giving it a symbolic name like COMMA. So everything appearing inside quotes in the grammar are *terminal* symbols just as `NEWLINE` is.

To finish, we need to designate a special symbol that will be the starting point of the grammar. In case of Python there are three starting points in its grammar:

```
single_input: NEWLINE | simple_stmt | compound_stmt NEWLINE
file_input: (NEWLINE | stmt)* ENDMARKER
eval_input: testlist NEWLINE* ENDMARKER
```

Those represent respectively a single statement, a full program and an expression.

We can also see a new special character here | which is used to represent alternatives: the `single_input` rule says that a symbol `single_input` is *either* a `NEWLINE` or a `simple_stmt` or the sequence `compound_stmt` and `NEWLINE`

4.1.2 Classification of Grammars

It is a good moment now to explain what a context-free grammar is (for more detail, see http://en.wikipedia.org/wiki/Context-free_grammar). We've seen a grammar has rules of the form:

```
S: a b c
```

and a grammar is said to be context free if the set of rules it contains can always be applied without conditions.

In fact grammars can be categorized in terms of expressivity (or how broad is the set of language they can represent) and complexity of parsing (how complex is the algorithm used to prove a program is correct according to the grammar). For example a language is context free if and only if it can be accepted by a non-deterministic push-down automata.

Chomsky organizes grammars in four types:

- type 0: unrestricted

- type 1: context-sensitive

- type 2: context-free

- type 3: regular

All context-free grammars can be analysed using an algorithm known as an *Earley parser*. This algorithm computes a grammar tree in $O(N^3)$ operations, N being the number of tokens.

N^3 can become very large and most programming languages are designed so that their grammar can be parsed with more efficient algorithms such as *LR* or *LL* parsers.

Python's grammar is mostly LL(1) which means that a parser can determine which rule to apply next by looking at the next token only. An LL(2) would require to look at the next 2 tokens. However some rules require more lookahead or need to be rewritten so that they actually become LL(1).

4.2 Brief Description of CPython's Implementation

In CPython, the grammar analyser is produced by reading the file `Grammar` and turning it into an automata based recognizer. The main *C* file of this program is `pgen.c`.

From `pgen.c`'s comments, the generation of the grammar is done as follows:

- First consider each rule as a regular expression and turn them into an NFA (Non-deterministic Finite Automaton).

- Then convert the NFA into a DFA (Deterministic Finite Automaton) - an operation proven to be always possible by increasing the number of states.

- The last step is to assemble the DFAs and optimize by reducing the number of states.

4.3 Syntax Tree Versus Abstract Syntax Tree

The last part of this introduction will explain what exactly the output of a grammar parser is. In fact parsing a program consists of producing a parse tree which is the tree of rules that needs to be applied to prove that the program is correct according to the grammar.

Here is a simple grammar for simple expressions:

```
expr: prod ('+'|'-') prod
prod: term | ( term ('*'|'/') term )
term: VARIABLE | NUMBER
```

With `VARIABLE` or `NUMBER` being *terminal* symbols matched by the tokenizer. For example we'll assume that they are recognized by the following regular expressions:

```
VARIABLE: [a-zA-Z]
NUMBER: [0-9]+
```

This very simple grammar can recognize expressions such as:

```
a+1
a*2+4*4
b-a/2
```

4.3.1 Parsing Algorithms

Let's take the last one -- "b-a/2" -- and show the analysis:

1. first `b` is tokenized as a `VARIABLE`
2. the `VARIABLE` is recognized as a `term`
3. a `term` being the first alternative for `prod` is recognized as a `prod`
4. with `prod` we have matched the first part of `expr`
5. according to `expr` the next token needs to be either `'+'` or `'-'`
6. `'-'` is matched and we need to recognize another `prod` symbol
7. a `prod` symbol starts with a `term` in both parts of its alternative so we expect a `term`
8. a `term` is either a `VARIABLE` or a `NUMBER` and the next token being `'a'`, we found a `VARIABLE`
9. we have found a complete `term` but at this point the rule for `prod` is ambiguous
10. assuming we know how to solve the ambiguity the next token being `'/'` we can advance in the recognition of the second alternative of `prod`
11. the next token being `NUMBER` we can complete a `term` and the `prod` rule which in turn finishes the rule `expr`

In the previous analysis we actually used two different methods:

- steps 1. through 4. is a bottom up analysis where by looking at a token we can deduce the rule to be matched and either *shift* to the next part of a rule or *reduce* a group of rules recognized as another rule. This is the principle of the algorithm used in *LR* parsers.

- steps 5. to 11. are a top down analysis where we start from the rule to be analysed and try to match subrules recursively. Such parsers are known as *recursive descent parsers*. A naive implementation of the parser will explore all possibilities recursively, eventually failing recognition and exploring other branches until the full list of tokens have been matched while attaining the end of the *goal* rule.

We also encountered an ambiguity while parsing top-down. This is a point where a recursive descent parser needs to explore two *matching* alternatives and decide which one to use based on the rules above it.

Usually this kind of ambiguities can be removed by rewriting the ambiguous part or using a better algorithm.

We can start by rewriting the rules like that:

```
expr: prod ('+'|'-') prod
prod: term [ ('*'|'/') term ]
term: VARIABLE | NUMBER
```

Now, if our algorithm looks at the next token to decide which rules can be applied, it will realize that the `'*'` can only be matched as part of the `prod` rule because following `prod`, we can only see `'+'`, `'-'` or the End Of Input.

4.3.2 Syntax Trees

We can represent the previous analysis with a functional expression:

```
expr( prod( term(VARIABLE('a') ) ),
      '-',
      prod( term( VARIABLE('a')),
                  '/',
                  term(NUMBER('2'))
          )
    )
```

This tree is called a syntax tree. Each rule is represented as a function of its matching subrules.

In the following we will also talk about Abstract Syntax Trees, which is a representation of the same program that is closer to the semantics of the program.

The following is a possible AST representation of our expression:

```
Substraction( Variable('a'), Division( Variable('a'), Number(2) ) )
```

5   History

Looking back at the evolution of the compiler/parser subsystem of PyPy, we can identify three major steps or *versions*:

First version: The first implementation used a handwritten tokenizer using regular expressions. The grammar parser's original design was not much different but it produced a tree of nested tuples that was in turn passed on to the CPython's original compiler package.

Second version: The second step had two objectives:

- Make the tokenizer translatable: to do this we reused Jonathan David Riehl's automata parser which was much easier to convert to RPython than the original tokenizer (at that time the regular expression module was not written in RPython yet).
- Make the parser translatable: this was mostly finished already with the caveat that the output data structure (a tree of tuples) could not be built with RPython. Thus we implemented a special builder that produced the equivalent structure as a tree of instances. The not-yet-translatable compiler was then taking care of converting this structure back into a tree of tuples.

Third version: This version was motivated by two observations:

- The Transformer class of the original CPython's implementation was buggy, complex and it would have been difficult to translate it to RPython. This class is responsible for turning the tree of tuples into an AST.
- The design of the parser made it possible to build the AST directly which is both an optimization and a way to avoid reusing a big hairy messy piece of code.

5.1   Tokenizer

The tokenizer went through three major versions, as described below.

5.1.1   A Regular Expression Based Tokenizer

This was the initial version. It was done quickly by translating directly the original C code into Python. We used regular expressions for optimization reasons since the original code parses the source one character at a time.

5.1.2   DFA Based Tokenizer

Another parser was developed more or less in parallel, mainly by Jonathan David Riehl with a different approach, especially the tokenizer which is based on Deterministic Finite Automata. The automata are defined with simple dictionaries with characters as keys, and integers as values. The integers represent a state in the automata, and the characters represent the transitions. The DFA is generated automatically, and can be regenerated at anytime, if needed, using a Python script.

We decided to use this tokenizer instead of the regexp-based one because the DFA-based tokenizer uses structures very close to what is allowed in RPython, and at the time regular expressions could not be used at interpreter level.

5.1.3   Translatable Version

The third and last version of the tokenizer only consisted of a few modifications to make the DFA-based tokenizer fully translatable and annotable.

5.2   Parser

All successive versions retained the same fundamental architecture based on several design patterns (composite, interpreter, builder, visitor).

Successive versions were implemented with different builders

5.3   Builder

There have been three major versions of the builder. At first the idea was to be able to produce exactly the same output as CPython's parser in order to be able to use CPython's compiler package.

CPython's parser module produces a syntax tree. The first version of the builder therefore built a syntax tree out of the tokens and used a second pass to visit the syntax tree and produced a tree of tuples that was suitable to feed the compiler.

It appeared that the second pass, used to build tuples out of a syntax tree, was not necessary because the syntax tree and the tuples were two nearly identical structures. The second version of the builder, also known as `tuplebuilder` was able to directly build the tuple tree from the token source.

This version of the builder was then made translatable and annotable to be integrated in version 0.6 of PyPy. Since the tuple tree is not RPython, the output was slightly modified to use only RPython structures and therefore is translatable.

The third and last version of the builder is the `AstBuidler`. This builder now directly transforms tokens into an AST. This allows to get rid of the `transformer` step that was needed by CPython's compiler package and which was used to transform the tree of tuples into the AST and was quite hard to make translatable.

5.4 Compiler

CPython provides a compiler package written in Python in its standard library. We decided to use it, at least as a basis, instead of reimplementing the whole thing from scratch. It is important to note that CPython does not use this package to compile Python source code, but uses its own C version of the compiler.

We first used the CPython's `compile` builtin function which uses the internal C version of the compiler, before starting to use a simple wrapper around the CPython's compiler package. This version used the output of our first parser.

The second version of the compiler used in PyPy was based on the `TupleBuilder` version of the parser. At this stage we used the compiler at application-level to transform our specific RPython structure into the nested tuples structure expected by the `transformer` step.

We then started to fix a lot of bugs found in the compiler package in order to pass the compliance tests for the 0.7 release of PyPy.

The third and current version of the compiler is a 50% rewrite of the compiler package. It is based on the `AstBuilder`, thus completely dropping the `transformer` step. This version also fixes a lot of bugs.

Bugfixes implemented in the third version of the compiler have been backported to the second version in order to provide a patch for the CPython's compiler package.

6 Architecture

6.1 Overview

The parsing process is distributed between three main actors: the `Tokenizer`, the `Parser` which is a set of `GrammarElements`, and the `Builder`.

6.1.1 Implementation

Here's a short description of the parser package:

- `grammar.py`: contains the definition of grammar elements related classes, especially `Sequence`, `Alternative` and `Token`. Python's grammar and the grammar's grammar are expressed in terms of instances of these classes. This module also contains the `build_first_sets()` function that builds the list of acceptable *first tokens* for each rule in the grammar.
- `ebnflexer.py`: the grammar's grammar tokenizer, based on regular expressions
- `ebnfparse.py`: contains the visitor and the builder that builds the grammar's grammar

- `pysymbol.py`: which is a replacement for the CPython's symbol module
- `pytoken.py`: which is a replacement for the CPython's token module
- `automata.py`: simple library for deterministic finite automata
- `pytokenize.py`: the generated automata definition for the Python source tokenizer
- `pythonlexer.py`: Python source's tokenizer, based on a DFA definition
- `pythonparse.py`: contains the function that builds Python's grammar
- `astbuilder.py`: contains the AST builder implementation (builder, version3)
- `tuplebuilder.py`: contains the nested tuple builder implementation (builder, version 2)
- `pythonutil.py`: a helper and facade module that provides high-level operations to build AST or tuples from Python sources.

The `pytokenize`, `pythonlexer` and `automata` modules were adapted from Jonathan David Riehl's Python parser.

6.1.2 Tokenizer

Our parser uses two distinct tokenizers: a tokenizer for the Grammar text file, and a tokenizer for the Python source code. The two share the same base class which is `TokenSource`. The token source is accessed through a simple iterator-like interface:

- `next()`: returns the next token in the source

Since Python's grammar is not strictly LL(1), we still need a backtracking mechanism for some tokenizer. To implement it, a source tokenizer needs to provide the following methods:

- `context()`: returns the current state of the token source
- `restore(context)`: restores a previous context. This function is used for backtracking.

A context object is a private object whose content depends on the implementation of a token source (See the Memento design pattern).

In the case of the Python tokenizer `context` objects are just indexes in the list of already recognized tokens.

6.1.3 Grammar

Each rule of the grammar is expressed in terms of instances of `GrammarElement`. We have four subclasses of `GrammarElement`:

- `Sequence` which is used to represent sequences, as in `S -> A B C`
- `Alternative` which is used to represent alternatives as in `S -> A | B | C`

- `KleeneStar` to express possible repetitions of a rule as in `S -> (A B)+`. The name "Kleene star" comes from the EBNF notation * which expresses 0 or more repetitions of the previous symbol. Here the class `KleeneStar` is used to describe any kind of repetition depending on its parameters. For example - `?` is `KleeneStar(min=0, max=1)` - `{2,5}` is `KleeneStar(min=2, max=5)` - `+` is `KleeneStar(min=1,max=-1)`
- `Token` to match a specific token

Those classes are a typical example of the `Composite` design pattern. They all share the same base class, and they hold references to other instances of the same type (i.e. a rule is itself a composition of several subrules).

Each `GrammarElement` subclass has a `match()` method whose role is to check if the next token, or the next set of tokens, matches the represented rule. The tokens are retrieved using the `tokenizer`.

A trivial example would be the `match()` method of the `Token` class. Its implementation consists of peeking the next token from the `tokenizer`, and checking the equality between the token it represents and the peeked one. If they are different, the rule does not match, if they are equal, the rule matches.

A slightly more involved example is `Alternative`'s match() method. It has to check if one of its alternate subrules matches the next set of tokens. The alternative is matched if any of the subrules is matched. This example also illustrates the recursive-descent aspect of the parser because each of the alternate subrules will itself have to check if it matches or not, and so on.

When a rule matches, it calls the `builder`'s corresponding method (`builder.sequence()` if the matched rule is a sequence, `builder.alternative()` if it is an alternative and so on). The `builder` is then responsible for building an appropriate object, such as the matching AST node.

The whole process is started by actually getting the toplevel `GrammarElement` that represents the toplevel rule (`file_input`, `single_input` or `eval_input` for the Python's grammar), and calling the `match()` method, passing the `tokenizer` and the `builder` as parameters.

The three parts we just described are really independent, and that's one of the reasons why our parser is so flexible. Being able to cleanly separate these three actions allowed us to have a working parser quickly, and to refine it later, with successive versions of the builder for example.

GrammarElement

+ symbols
+ codename
+ args
+ first_set
+ first_set_complete
# _trace

+ __init__()
+ is_root()
+ match()
+ _debug_display()
+ _match()
+ parse()
+ __str__()
+ __repr__()
+ display()
+ debug_return()
+ calc_first_set()
+ match_first_set()
+ in_first_set()
+ reorder_rule()
+ validate()

Token

+ value
+ first_set

+ __init__()
+ match()
+ display()
+ match_token()
+ __eq__()
+ calc_first_set()
+ validate()

Sequence

+ args
+ first_set

+ __init__()
+ _match()
+ display()
+ calc_first_set()
+ validate()

Alternative

+ args
# _reordered
+ first_set

+ __init__()
+ _match()
+ display()
+ calc_first_set()
+ reorder_rule()
+ validate()

KleenStar

+ args
+ min
+ max
+ star
+ first_set

+ __init__()
+ _match()
+ display()
+ calc_first_set()
+ validate()

Figure: subclasses of GrammarElements

6.1.4 Parsers

The parser is a tree of grammar rules represented by instances of `GrammarElement`. EBNF grammar rules are decomposed as a tree of objects. Looking at a grammar rule, one can see that it is composed of a set of four basic subrules. In the following example we have all the four of them:

```
S <- A '+' (C | D) +
```

The previous line says S is a sequence of symbol A, token '+', and a subrule which matches one or more of an alternative between symbol C and symbol D.

Thus the four basic grammar rule types are:
- sequence
- alternative
- multiplicity (called kleen star after the * multiplicity type)
- token

The four types are represented by a class in pyparser/grammar.py (Sequence, Alternative, KleeneStar, Token) all classes have a `match()` method accepting a source (the tokenizer) and a builder (an object responsible for building something out of the grammar).

Here's a basic exemple and how the grammar is represented:

```
S <- A ('+'|'-') A
A <- V ( ('*'|'/') V )*
V <- 'x' | 'y'
```

In python:

```
V = Alternative( Token('x'), Token('y') )
A = Sequence( V,
        KleeneStar(
            Sequence(
                Alternative( Token('*'), Token('/') ), V
                    )
                )
            )
S = Sequence( A, Alternative( Token('+'), Token('-') ), A )
```

6.2 The Grammar's Grammar Parser

This section explains how we create a tree of grammar elements from the text file `Grammar` which describes Python's grammar for each version.

6.2.1 Principles

The python grammar is built at startup from the pristine CPython grammar file: `Grammar`.

Building the correct CPython's grammar is a two step process:

1. Create the grammar's grammar.
2. Build Python's grammar by feeding a specific builder to the grammar's grammar.

As the syntax is likely to change between different versions of Python, it is important for PyPy to have an automated way to build a correct grammar object depending on the version of Python we want to use. Of course there is still work to do to interpret correctly new constructs but at least the parsing will be done automatically.

The Python grammar is defined in a simple text file. This text file has a well-defined frozen syntax itself (i.e never changes across versions of Python). Therefore, for step one, we have defined once and for all the grammar of this text file in terms of our Grammar objects (GrammarElement subclasses).

The second step is achieved by providing a simple tokenizer for the grammar's grammar and a specific builder that is responsible to create appropriate GrammarElement's instances according to the matched rule.

6.2.2 Building the Grammar's Grammar

Here's the description of the grammar's grammar:

```
grammar: rule+
rule: SYMDEF alternative

alternative: sequence ( '|' sequence )+
star: '*' | '+'
sequence: (SYMBOL star? | STRING | option | group star? )+
option: '[' alternative ']'
group: '(' alternative ')' star?
```

And below is the code that generates the grammar's parser:

```
def grammar_grammar():
    """Builds the grammar for the grammar file
    """
    global rules, sym_map
    S = g_add_symbol
    # star: '*' | '+'
    star        = Alternative( S("star"), [Token(S('*')), Token(S('+'))] )
    star_opt    = KleeneStar  ( S("star_opt"), 0, 1, rule=star )

    # rule: SYMBOL ':' alternative
    symbol      = Sequence(  S("symbol"), [Token(S('SYMBOL')), star_opt] )
    symboldef   = Token(     S("SYMDEF") )
    alternative = Sequence(  S("alternative"), [])
    rule        = Sequence(  S("rule"), [symboldef, alternative] )

    # grammar: rule+
    grammar     = KleeneStar(   S("grammar"), _min=1, rule=rule )

    # alternative: sequence ( '|' sequence )*
    sequence      = KleeneStar( S("sequence"), 1 )
    seq_cont_list = Sequence(  S("seq_cont_list"),
                               [Token(S('|')), sequence] )
    sequence_cont = KleeneStar( S("sequence_cont"),0, rule=seq_cont_list )

    alternative.args = [ sequence, sequence_cont ]

    # option: '[' alternative ']'
    option      = Sequence(S("option"),
                           [Token(S('[')), alternative, Token(S(']'))])

    # group: '(' alternative ')'
    group       = Sequence(   S("group"),
                              [Token(S('(')), alternative, Token(S(')')),
                               star_opt] )

    # sequence: (SYMBOL | STRING | option | group )+
    string = Token(S('STRING'))
```

```
alt            = Alternative( S("sequence_alt"),
                              [symbol, string, option, group] )
sequence.args = [ alt ]


rules = [ star, star_opt, symbol, alternative, rule, grammar, sequence,
          seq_cont_list, sequence_cont, option, group, alt ]
build_first_sets( rules )
return grammar
```

6.2.3   The Grammar Tokenizer

The grammar Tokenizer is very simple and implemented in `ebnflexer.py`.

It knows only 5 types of tokens:

- `EOF`: end of file
- `SYMDEF`: a symbol definition e.g. `file_input:`
- `STRING`: a simple string "'xxx'"
- `SYMBOL`: a rule symbol usually appeary right of a `SYMDEF`
- simple tokens: '(', ')', '(' ,')', '*', '+', '|'

6.2.4   The Generated Syntax Tree

The grammar builder is the default syntax tree builder `BaseGrammarBuilder`. This builder produces the syntax tree of the file (not the AST!). for example the rule:

```
testlist1: test (',' test)*
```

is turned into the tree:

```
SyntaxNode('rule', SYMDEF('testlist1:'),
  SyntaxNode('alternative',
    SyntaxNode('sequence',
      SYMBOL('test'),
      SyntaxNode('group', STRING('('),
        SyntaxNode('alternative',
          SyntaxNode('sequence', STRING(','), SYMBOL(test') )
        ),
        STRING(')'),
      ),
      SyntaxNode('star', STRING('*') )
    )
  )
)
```

The root of this tree is a `SyntaxNode('grammar', ...)` where the `...` is a list containing `SyntaxNode('rule',...)`.

6.2.5   Turning the Syntax Tree into a Grammar

Once we obtain a syntax tree for the grammar file we need to turn it into a graph of instances of `GrammarElement`. This step is implemented in `ebnfparse.py` by the class `EBNFVisitor` and is a bit more complex.

The visitor is applied upon the syntax tree. It has a `visit_` method to process each type of node found in the tree.

The visitor uses a stack based algorithm to build the graph representation of each rule. The actual linking between the rule symbol and the rule objects is done in a second pass since the order of the rules in the file is free.

For example the processing of the subtree:

```
SyntaxNode('sequence', STRING(','), SYMBOL('test') )
```

gives the subgraph:

```
Sequence(':unnamed_node', Token(','), test )
```

where `test` is the actual reference to a `GrammarElement` representing the rule for `test`.

What we actually described here is the transformation of a syntax tree (the tree of `SyntaxNode`) into an abstract syntax tree (in fact a graph of subclasses of `GrammarElement`)

Of course some simplifications/optimizations occur in the process when for example we encounter:

```
SyntaxNode( 'alternative', SyntaxNode('sequence',...) )
```

In that case we won't generate an `Alternative` node with only a single alternative branch.

A last important thing to mention is the role of anonymous subrules. Consider the following rule:

```
and_test: not_test ('and' not_test)*
```

We have already explained that it is parsed and transformed in a way to be expressed in terms of `GrammarElement`s instances. But to be able to do that, we must create an anonymous grammar rule for the `('and' not_test)*` part:

```
KleeneStar(Token('and'), not_test)
```

Notice that the sequence generated above has a name `:unnamed_node`. In the Python grammar parser we use the convention that if node's name starts with a ':' it is an anonymous node. For debugging purposes the name after the ':' is the name of the closest *named* parent rule.

6.3 Backtracking Support

6.3.1 The Python Grammar Recognizer

In the previous section we explained how this grammar recognizer is built automatically from a description in a file.

Such a graph really is an LL(k) parser, because every grammar element provides the `match()` method seen earlier.

Backtracking support is implemented by the source (tokenizer) and builder if needed (e.g. grammar's grammar is LL(1) thus the builder does not need to support backtracking) whereas CPython's grammar builders and tokenizer need backtracking because the Python grammar is not LL(1)

6.3.2 The Tokenizer

The tokenizer was kept very simple because most of the logic is done by the grammar (i.e. a keyword or symbol is determined at parsing time). It basically accepts a string as input and provides tokens through a `next()` and a `peek()` method. The tokenizer is implemented as a finite automaton like lex.

6.4 The Builders

Figure: AbstractBuilders class hierarchy and collaboration

6.4.1 TupleBuilder (version 2.1)

CPython's `transformer` module is responsible for taking the parser's output and transforming it into an Abstract Syntax Tree. This is a two step process:
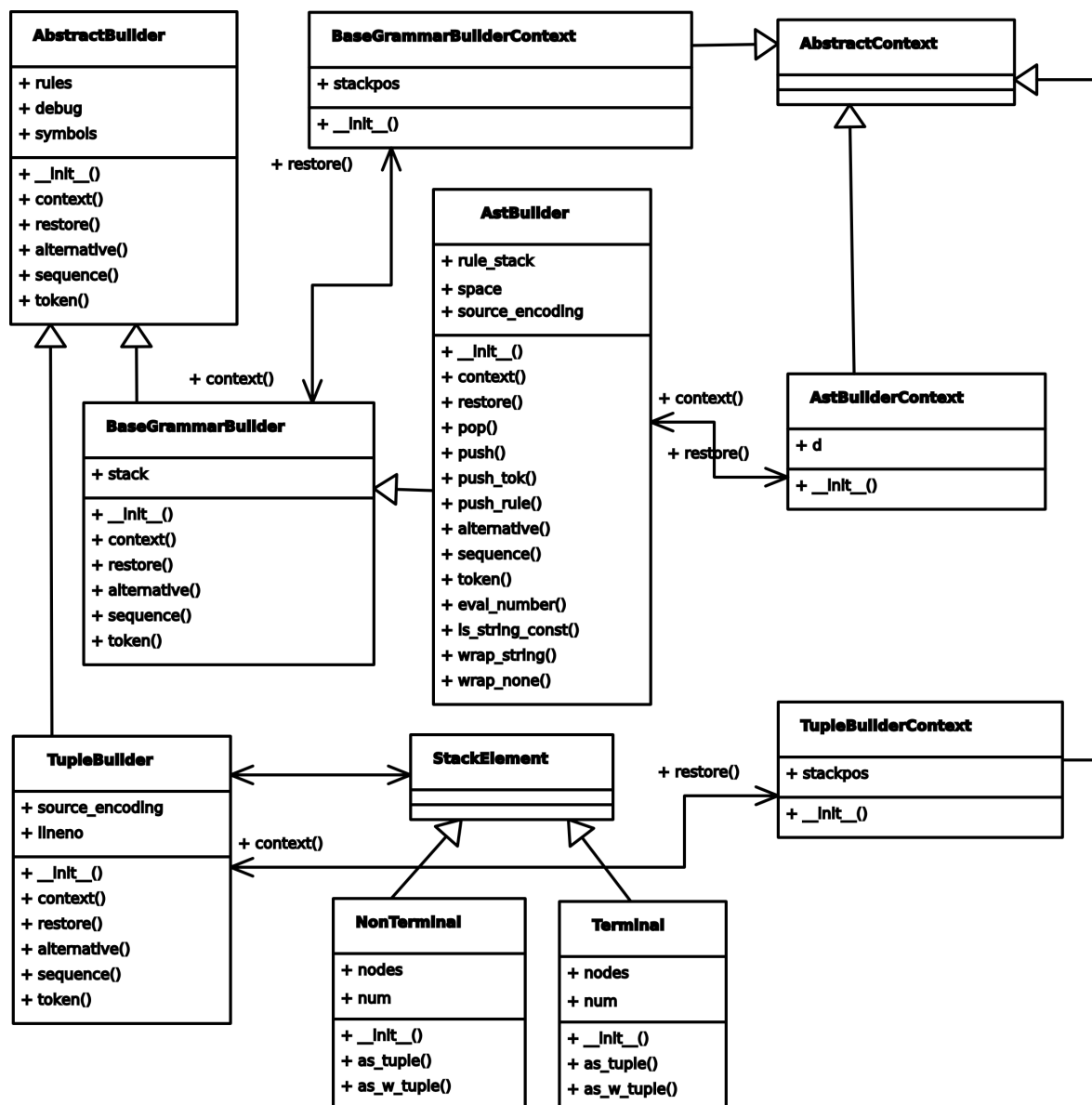
1. Take the parser's output (i.e. a Syntax Tree) and transform it into nested tuples which are a tuple representation of the tree.
2. Visit this tree, and create the matching Abstract Syntax Tree.

The tuple builder we implemented directly produces nested tuples that are usable as the second step's input.

The tuplebuilder holds a stack of tuples. The 4 kinds of grammar elements are transformed as follows:

1. Each time a token is matched, a tuple `(token_code, token_value, lineno)` is pushed on the stack.
2. When an alternative is matched, nothing is done if the alternative is not a *named* rule. Otherwise a tuple is created with `(rule code, top of the stack, lineno)`.
3. A KleeneStar is treated as a sequence, when a KleeneStar is encountered, the `Builder.sequence` method is called with the sequence of elements matched.
4. Finally, when a sequence is matched, it pops the Nth top elements of the stack, where N is the number of elements in the sequence, and pushes a nested tuple built from those N elements.

6.4.2 TupleBuilder (version 2.2)

The major problem of the approach described previously is that nested tuples of variable length cannot be translated with PyPy. Therefore in order to make PyPy's parser annotatable we added a level of abstraction to encapsulate the tuples. This new tuple builder now holds a stack of `terminal` elements (for tokens) and `non terminal` elements (for sequences and alternatives) which both share a common `StackElement` base class. Each stack element owns itself a list of other stack elements. This composite structure reflects the original structure of nested tuples, but in an RPythonic way:

- the builder's stack now is only composed of instances of `StackElement`
- StackElement's instances own lists of StackElement instances.

But here we still need the compiler package to generate the Abstract Syntax Tree, so we provide a way to transform our new `stack elements` into a nested tuple, and then pass this tuple to the compiler package.

This builder implementation provides a fully translatable parser, but still requires the nested tuples generation step to be able to produce the AST.

6.4.3 AstBuilder

In order to get rid of the nested tuples structure, we have implemented a new builder which directly generates the Abstract Syntax Tree from the tokens. The two major benefits are that we only use RPython structures and that we can get rid of the tuples to AST transformation step.

The `astbuilder` module defines specific AST nodes by extending the base `Node` class provided by the compiler package.

For example, the `TokenObject` class is defined to wrap simple tokens and store the token's name, value and line number. This class, as well as all the node classes defined in the `astbuilder` module, are only here to build temporary objects that share the same base class

as the *real* AST nodes, and therefore help the translation process, but none of their instances will be part of the final AST representation.

The basic idea behind the AstBuilder is to push AST nodes (*real* or *temporary* as stated above) on the builder's stack until we are matching a rule for which the module defines a *reducing* function. Reducing functions act like node factories. They all have exactly the same prototype, which is:

```
def build_some_node(builder, n):
    # 1. pops the n topmost elements of the builder's stack
    # 2. builds as appropriate AST node and pushes it on the
    #    builder's stack
    # does not return anything
```

When a rule is matched but does not have a corresponding reducing function, a `RuleObject` instance is pushed on the stack. RuleObject is a class, just like the TokenObject class we mentioned earlier, which extends the base AST node class.

The only information we need to store is the number of tokens that this rule gathers because the next reducing function will have to pop all those elements.

For example, for a sequence of 3 elements for which there is no corresponding reducing function, the builder will push a RuleObject's instance that will remember the number of tokens matched by this rule, that is 3 in this case.

Then when popping elements from the stack, the next reducing function will find this RuleObject instance, and will have to pop the next 3 elements that were matched by this rule.

A simple real example of a reducing function is the `Yield` one which is called when the matched rule is the `yield_stmt` one (defined in the grammar):

```
# The yield_stmt rule is: yield_stmt: 'yield' testlist
def build_yield_stmt(builder, nb):
    atoms = get_atoms(builder, nb)
    yield_node = ast.Yield(atoms[1], atoms[0].lineno)
    builder.push(yield_node)
```

The `get_atoms()` function pops the appropriate number of elements from the builder's stack and returns this list. It then builds the Yield's AST node, and pushes it on the top on the stack. Here is the get_atoms() function's algorithm:

```
def get_atoms(stack, N):
    atoms = []
    while N != 0:
        elt = stack.pop()
        if type(elt) is RuleObject:
            N = N + elt.matched_elements_count
        else:
            atoms.append(elt)
            N -= 1
```

At the end of the build process, the stack is completely reduced, and there's only one node left which is the final AST tree itself.

An important difference between the ways AstBuilder and CPython's compiler build the AST is that ASTBuilder builds the tree from bottom to top, and the transformer does it the other way.

Building from bottom to top is a bit harder because we are sometimes missing information when building the AST node.

For example, the corresponding ast node for the `a.b` expression is not the same depending on if it is a left or right part of an assignment (`c = a.b` or `a.b = c`). When building from top to bottom, as the transformer does, you detect that you are building an assignment expression before you have to produce the AST node for the left and right parts. The AstBuilder, which does it the other way, starts by reducing `a.b` before reducing the whole assignment, and therefore it has no clue that it is part of an assignment at this time. To build the correct AST, we therefore have to modify nodes afterwards or to defer node instantiation.

6.5   Compiler

Here the idea was to rely as much as possible on the existing code in the CPython's compiler package which generates bytecode by visiting the Abstract Syntax Tree. Unfortunately, some big parts of this package are not RPython, and this compiler is far from being 100% compatible with the original version written in C. However, we kept the general algorithm and the main architecture provided by this package.

6.5.1   Existing Design of the Compiler

The compiler package has three main parts:

- a transformer that turns syntax trees into Abstract Syntax Trees
- the core compiler classes that produce a flowgraph of bytecode
- a bytecode *assembler* that produces Python Code objects by assembling the blocks of the flow graph

6.5.2   The Transformer

As stated earlier the transformer is used to transform nested tuples into Abstract Syntax Tree. This part of the CPython's compiler package is very difficult to turn into RPython because it takes and manipulates non RPython structures. The tuple builders versions need it to build the AST.

The transformer has been dropped in the last version since it used a data structure as input that cannot be represented by PyPy's type analyser. The reason is that its input is a tree of tuples of the form:

```
('rulename', rule argument or token, line number if requested)
```

PyPy's typer recognizes either a list of objects with the same types or a tuple of objects with different **but defined** types. A list cannot be used here because we have a tuple of (string, tuple(ofxxx), int). Thus we need a tuple. Representing the second argument could be possible if it were not for the *rule or token*, because a token tuple is different from a rule tuple the second slot of the tuple cannot be typed properly.

Furthermore, changing this representation implied changing most of the transformer code so the path we chose was to produce AST directly with the AST builder described earlier.

6.5.3   The Compiler

The compiler itself is built around several visitor objects. There are three entry points depending on the target rule of the grammar used to produce the AST tree. These entry points are used to produce either a Module, or a Code Object.

The selection of the variable scope analyser, which is also a visitor, is also depending on the entry point.

Then most visitors inherit from a base visitor `CodeGenerator`. This visitor handles most AST nodes, like those used for representing statements.

The variations provided by derived classes determine the creation of new code objects through the use of a different assembler flowgraph. They also help determining the scopes of variables appearing in the lower nodes.

The following subclasses exists:

- ModuleCodeGenerator
- ExpressionCodeGenerator
- InteractiveCodeGenerator
- AbstractFunctionCode
- FunctionCodeGenerator
- GenExprCodeGenerator
- AbstractClassCode
- ClassCodeGenerator

CodeGenerator

ModuleCodeGenerator

+ futures

+ __init__()
+ get_module()

InteractiveCodeGenerator

+ futures

+ __init__()
+ get_module()
+ visitDiscard()

ExpressionCodeGenerator

+ futures

+ __init__()
+ get_module()

AbstractFunctionCode

+ __init__()
+ get_module()
+ finish()
+ generateArgUnpack()
+ unpackSequence()

FunctionCodeGenerator

+ scope
+ localsfullyknown
+ scopeambiguity

GenExprCodeGenerator

+ scope
+ localsfullyknown
+ scopeambiguity

AbstractClassCode

+ __init__()
+ get_module()
+ finish()

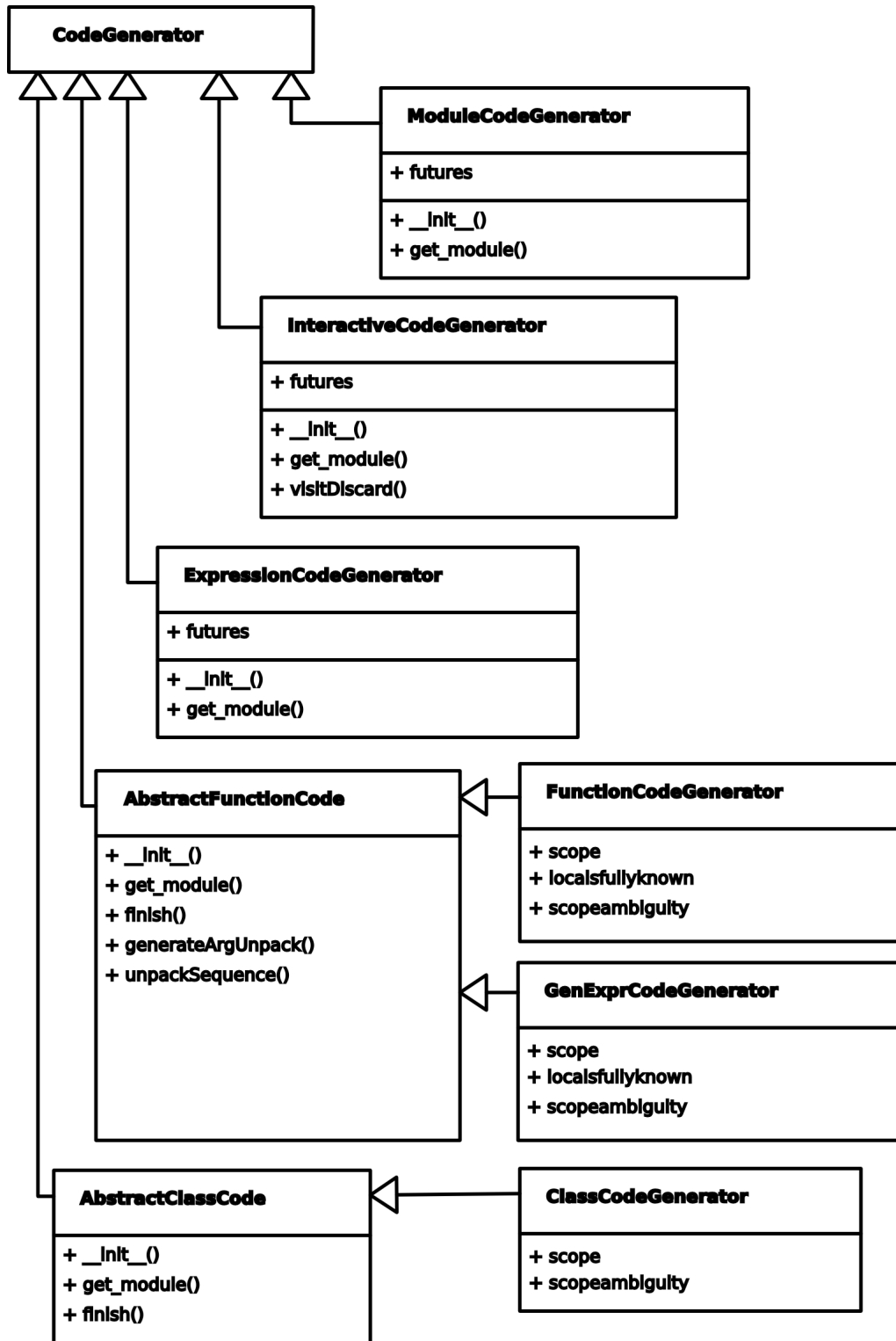**ClassCodeGenerator**

+ scope
+ scopeambiguity

Figure: Visitor hierarchy from `pycodegen.py`

6.5.4 RPython Visitor Pattern

We had to change the AST base classes in order to make the compiler part RPython, and we changed the way the tree was visited to use a real proper implementation of the visitor pattern.

The visitor pattern in Python can be implemented in a very dynamic way, and this was the case with the visitors of the compiler package. For example calling the visit method from a node can be made very generic with Python: if a node's class is Multiply, calling the `node.accept()` method with a argument of `visitor` will in turn call `visitor.visit(self)` on the visitor is sufficient because the `visit` method can look at the node's `__class__` attribute and dispatch on the appropriate `visit__Multiply` method. A sample visit implementation looks like:

```
def visit(self, node):
    func_name = "visit_%s" % node.__class__.__name__
    if hasattr(self, func_name):
        method = getattr(self, func_name)
        method(node)
```

This is far too dynamic for RPython to cope with, and needed to be rewritten in a more *standard* way. That is each AST node needs to override the accept method to call the specific visitor method directly. In our example:

```
class Multiply(ASTNode):
    def accept(self, visitor):
        visitor.visit_Multiply(self)
```

Fortunately, all nodes from the AST tree were already generated from a template file so this modification only involved changing the generator.

Some other points needed to be fixed the same way: in RPython a function cannot accept arguments with multiple types or a variable number of arguments. Those places needed more work because most of the time this feature was used to pass a state to deeper branches of the AST tree. The generic solution to this problem is to implement each state as a stack attribute of the visitor object and have the corresponding `visit_xxx` methods look at the top of this stack for the parameter.

6.5.5 Bug Fixing

A great effort was made to fix all the bugs found in the CPython's compiler package. Unfortunately, it is hard to have reliable tests that test the compiler in a deep and exhaustive way. Of course, it is still possible to compare generated bytecodes of the PyPy compiler and the CPython's compiler package ones, but since the latter has a lot of bugs in corner cases, it is not always valid to compare both generated bytecodes.

The solution of comparing PyPy's generated bytecodes and CPython's ones (using the *official* compiler written in C) is not that good a solution either, because the CPython's compiler does some optimizations for which we don't care for now. Therefore in addition to the lots of small snippets of Python source code that can be tested easily, we chose to consider our compiler operational as long as all the compliance tests pass.

6.5.6 Contributing the Bugfixes

We plan to backport the fixes we implemented into the official CPython's compiler package by providing a patch in the next few weeks.

6.6 Bytecode Generation

The main classes of the Bytecode Generator are `FlowGraph` and `PyFlowGraph`.

A flow graph maintains a graph of `Block` objects and provides methods for linking Blocks together. The flowgraph of bytecode follows the same principles as the flowgraphs described in D04.2. It is much simpler here as Blocks contain a series of bytecodes without branches. the *out edges* of a Block represent possible changes in the control flow of the bytecode while the *in edges* represent the links from other Blocks' *out edges*.

FlowGraph

+ space
+ current
# _debug

+ __init__()
+ startBlock()
+ nextBlock()
+ newBlock()
+ startExitBlock()
+ _enable_debug()
+ _disable_debug()
+ emit()
+ emitop_obj()
+ emitop_code()
+ emitop_int()
+ emitop_block()
+ emitop_name()
+ getBlocksInOrder()
+ fixupOrder()
+ fixupOrderHonorNext()
+ fixupOrderForward()
+ getBlocks()
+ getRoot()
+ getContainedGraphs()

BlockSet

+ elts

+ __init__()
+ __len__()
+ __contains__()
+ add()
+ elements()
+ has_elt()
+ remove()
+ copy()

+ blocks
+ inEdges
+ outEdges
+ exit

Block

+ insts
+ label
+ bid
+ space

+ __init__()
+ __repr__()
+ __str__()
+ emit()
+ getInstructions()
+ addInEdge()
+ addOutEdge()
+ addNext()
+ pruneNext()
+ get_children()
+ getContainedGraphs()

+ next

PyFlowGraph

+ __init__()
+ setDocstring()
+ setFlag()
+ checkFlag()
+ setFreeVars()
+ setCellVars()
+ getCode()
+ dump()
+ _max_depth()
+ computeStackDepth()
+ flattenGraph()
+ convertArgs()
+ sort_cellvars()
+ _lookupName()
+ _cmpConsts()
+ _lookupConst()
+ _convert_LOAD_CONST()
+ _convert_LOAD_FAST()
+ _convert_NAME()
+ _convert_DEREF()
+ _convert_LOAD_CLOSURE()
+ _convert_COMPARE_OP()
+ makeByteCode()
+ newCodeObject()
+ getConsts()

Figure: The flowgraph class diagram

In other word a block can be the target of several branches like in:

```
X = input()
if X:
    print X
X = 0
```

The block formed by the X=0 instruction has two *in edges*: one coming from the end of `x = input()` and one coming from the `print X` block. The `if X` line has two *out edges* depending on the outcome of the test. One points to `print X` and the other one points to `X = 0`.

While `FlowGraph` is a base class dealing only with generic flow graphs of Blocks, `PyFlowGraph` is a specialization that deals with Python bytecode, and can reassemble the Blocks into a Python `CodeObject`.

The method `flattenGraph` is the main part of `PyFlowGraph`. Its job is to *flatten* the graph by trying to put the Blocks in a list in a way that minimizes the number of jumps needed (mostly like a topological sort). Once the blocks are laid out, the jump points are computed and the argument to bytecode *jump* operations are replaced by the actual bytecode position.

A limitation of the algorithm is that long jumps (more than 65435 bytes) can only be made backward. The reason is that Python bytecodes only have 16 bits arguments, and larger arguments are encoded using two bytecodes (the first one being an *extended argument*). Adding such a bytecode while assembling the blocks would cause all forward references to changes invalidating all previous jumps. The price to implement this correctly is not worth it as:

1. CPython does not do it
2. There is almost never any code object of this size
3. A code object is produced for every function, thus having a very large code object means the function is very complex which is a sure sign of bad programming practice.

The remaining operation is the computation of the line number table which is a kind of compressed table encoding the number of bytes in the bytecode between two changes of line numbers and the difference between those line numbers. The table almost always needs two bytes, a special case is provided to encode gaps of more than 255 bytes.
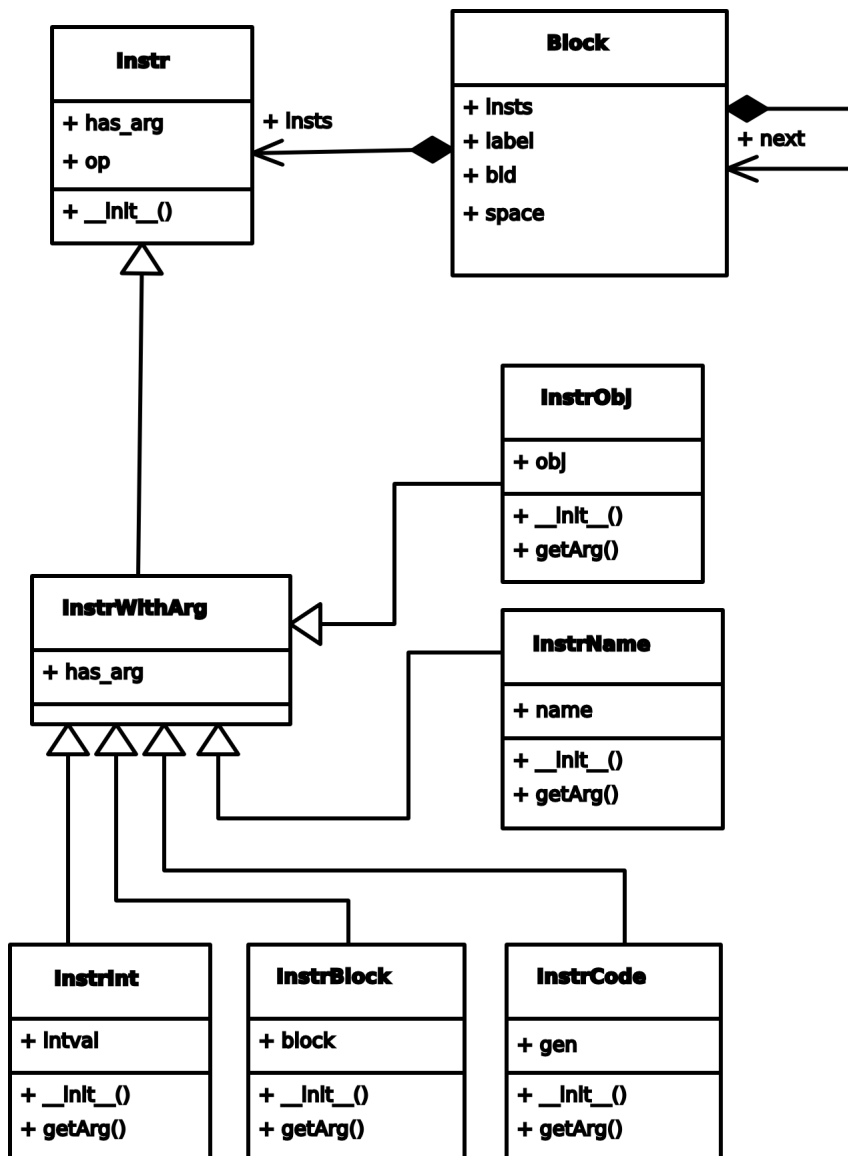
Figure: RPython considerations: In order to make the assembler RPython compliant, all instructions have been encapsulated in instances deriving from `Instr`

7 Future Evolutions

The parser and compiler were implemented in the most easy-to-understand and easy-to-extend way because it will be used as the basis of several experiences we want to do in the future, especially syntax experiments and modifications, optimizations and code transformation tools.

With our parser, it is of course possible to make small experiments to modify the Python's syntax statically. It is basically feasible by only changing the grammar text file (provided that we do not have to change the bytecode generation). But we now would like to investigate on how to modify the Python's syntax dynamically. For example, we would like to provide an `Oz` like

syntax for those who want to do logic and constraint programming. An idea would be to do that when the following statement would be seen:

```
import csp_syntax
```

In the rest of the module, we would then be able to use a mix of `Oz` and Python syntax. While we can imagine ways to do that with our parser and compiler architecture, it is of course still not trivial. We would have to provide ways to plug new AST factory functions in order to produce new AST nodes.

Extending the syntax also probably means defining new AST node types, which in turn means that we have to provide a way to easily extend the AST nodes, subclassing AST nodes from the Python interpreter, which also probably means being able to modify the compiler visitor so that the new AST nodes are properly recognized.

We could also use tree rewriting techniques to change AST trees before compilation, for optimizations purposes or code injections (for example, to implement ideas from aspect oriented programming or design by contract-style assertions).

One last point would be to extend the available bytecodes. This is certainly not trivial because it involves providing hooks into the evaluation loop and possibly adding new operations to the Object Spaces. The existing bytecode is generic enough that there is probably no need to do that dynamically. Adding bytecodes for optimization reasons can still be done.

These additions will be included in PyPy based on the requirements from the constraint programming and aspects workpackages (WP9 and WP10).

8   Glossary of Abbreviations

The following abbreviations may be used within this document:

8.1   Technical Abbreviations:

AST	Abstract Syntax Tree
CPython	The standard Python interpreter written in C. Generally known as "Python". Available from www.python.org.
codespeak	The name of the machine where the PyPy project is hosted.
docutils	The Python documentation utilities.
GenC backend	The backend for the PyPy translation toolsuite that generates C code.
GenLLVM backend	The backend for the PyPy translation toolsuite that generates LLVM code.
Graphviz	Graph visualisation software from AT&T.
Jython	A version of Python written in Java.
LLVM	Low Level Virtual Machine - a compiler infrastructure available from University of Illinois at Urbana-Champaign
LOC	Lines of code.
Object Space	A library providing objects and operations between them, available to the bytecode interpreter via a well-defined API.

Pygame	A Python extension library that wraps the Simple Direct-media Library - a cross-platform multimedia library designed to provide fast access to the graphics framebuffer and audio device.
ReST	reStructuredText, the plaintext markup system used by docutils.
RPython	Restricted Python; a less dynamic subset of Python in which PyPy is written.
Standard Interpreter	The subsystem of PyPy which implements the Python language. It is divided in two components: the bytecode interpreter, and the standard object space.
Standard Object Space	An object space which implements creation, access and modification of regular Python application level objects.

8.2 Partner Acronyms:

DFKI	Deutsches Forschungszentrum für künstliche Intelligenz
HHU	Heinrich Heine Universität Düsseldorf
Strakt	AB Strakt
Logilab	Logilab
CM	Change Maker
mer	merlinux GmbH
tis	Tismerysoft GmbH