THE UNIVERSITY OF QUEENSLAND

School of Information Technology and Electrical Engineering

TsPyC: A Programming Language Supporting Modular, Robust Extension

By

J. D. Bartlett

School of Information Technology and Electrical Engineering

The University of Queensland

Submitted for the degree of

Bachelor of Engineering (Honours)

in the field of Software Engineering

30 October 2009

Mr Joshua D. Bartlett
Arana Hills, Brisbane,
Q. 4054

30 October 2009

To the Head of School
School of Information Technology and Electrical Engineering,
The University of Queensland
St Lucia, Q. 4072

Dear Professor Bailes,

In accordance with the requirements of the degree of Bachelor of Engineering (Honours) in the field of Software Engineering, I submit the following thesis entitled

TsPyC: A Programming Language Supporting Modular, Robust Extension

This project was performed under the supervision of Professor Ian Hayes.

I declare that the work submitted in this thesis is my own, except as acknowledged in the text and footnotes, and has not been previously submitted for a degree at the University of Queensland or any other institution.

Yours sincerely,

Joshua D. Bartlett

Abstract

The aim of this project was to design and implement the programming language tsPyC (rhymes with "spicy"). This raises the obvious question, "Why do we need another programming language?" TsPyC is different because it gives programmers the flexibility to write robust, modular extensions which add new features to the language.

By writing language extensions, programmers can tailor the language to their domain-specific applications. For example, an extension module could contain definitions of matrix operations together with matrix-related language constructs to allow more readable tsPyC source code.

Programmers can write language extensions as Python modules, and have access to the full capabilities of the Python programming language. Because extensions are modular, they are self-contained and can easily be shared with other developers.

In order to make the extensions as robust as the standard features of the language, compile-time checking can be included in the extensions. For example, an extension may define data types which represent physical quantities with units (such as metres or seconds). This extension could include compile-time checks which prohibit operations such as addition or variable assignment when the units are not consistent. So attempting to add a distance to a time would result in a compile-time error being reported.

TsPyC generates native machine code as output, and uses C code as an intermediate step. The C code generator can be used separately from the rest of the tsPyC compiler, allowing it to be used in a wide range of different applications.

Acknowledgements

I would like to acknowledge the assistance of my supervisor, Professor Ian Hayes, in the completion of this thesis. He showed remarkable patience and trust, even at those times when he wasn't entirely sure what I was trying to achieve or why. I would also like to acknowledge the support of my family, who put up with me when I was busy and sometimes even stressed. I am especially thankful for the encouragement and support shown to me by my mother Helen and my fianceé Alicia (pr. /əˈliːsiə/). I am grateful also to my friends, particularly Jake Owen and Ashley Donaldson for their ongoing interest in the progress of this project. And finally, I must acknowledge the gracious provision of my God, without whom I would have been able neither to complete this thesis, nor even to exist.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

1.1 Motivation and Aims

The aim of this project was to design and develop the new programming language tsPyC (rhymes with "spicy"). The obvious question is "There are so many programming languages already—why do we need another one?" The main focus when developing tsPyC was to make the language flexible enough that programmers could add language features by writing robust, modular extensions.

Imagine that you plan to write a program in order to solve a problem in some domain. To solve this problem you may require a particular language feature. For instance, solving your problem may require the use of matrices, and you decide that the problem would be much easier to solve using a language which natively supports matrices and matrix operations. Or perhaps you're trying to put a satellite in orbit and you need to be able to convince yourself and others that your solution is going to work. As a step towards doing this, you would like a language which supports unit checking, just to make sure that you haven't accidentally assumed that a quantity is in seconds in one part of the code, but in minutes elsewhere.

Given a set of desired language features, there is generally a small number of options available to you. You can find a language which already has support for units and matrices. Such languages exist, but the more specific a set of language features you desire, the more difficult it becomes to find such a language. Alternatively, you could write your own domain specific language to help you solve the problem. This practice is not unheard of, particularly in situations where there are likely to be many problems to solve in a particular domain.

As a third alternative, you could use an existing programming language without support for units and matrices, but add the required functionality yourself. For example, you might define a class to represent a quantity with units. Instances of this class could store not only a value, but also a unit. Checking for consistency of units could then be performed at run-time. Solving problems in this way has several drawbacks. In the case of unit checking, the run-time nature of the checking means that you can only tell that you've made a mistake when actually running the program, and not during the build process. But more generally, it is likely that your intentions can be expressed more clearly in a language with native support for a particular feature, than in a language without.

TsPyC provides a solution to this problem by allowing a language feature to be defined in a modular extension. These extensions take the form of Python modules which contain instructions run by the compiler at the time that a given tsPyC program is compiled.

Having a flexible language which can be extended by programmers was a key aim of tsPyC. The other key aim was for tsPyC to compile to native machine code, and to be retargetable for different CPUs. The reason that it was considered important for tsPyC to output native machine code is that generally, the languages with the greatest flexibility and extensibility generate virtual machine code which must be interpreted. This means that programmers pay for flexibility with performance—executing code written in such languages is much less efficient than executing native machine code. This project set out to provide flexibility without sacrificing efficient run-time performance.

1.2 Background

1.2.1 Extensible Programming

In the 1960s and early 1970s, much work was done on the concept of *extensible programming*. The concept revolved around the idea of providing mechanisms by which the core features of a programming language could be supplemented, often by making use of some kind of *meta-language* in which the definition of the base language was expressed. This work on extensible programming often focused particularly on the abilities to define macros and to adapt the grammar of the language. For a 1975 review of the topic of extensible programming, see [15].

More recently, there has been renewed interest in the concept of extensible programming. One advocate of this concept is Gregory Wilson of the University of Toronto, who argues that next-generation programming languages should have the ability to be customised using plug-ins, should allow programmers to extend their syntax, and should store programs as XML documents so that data and meta-data can be represented and processed in a uniform way. Wilson claims that "these innovations will likely change programming as profoundly as structured languages did in the 1970s, objects in the 1980s, and components and reflection in the 1990s." [16]

While it is important to know of other work in areas related to the current project, it should be noted that the tsPyC language falls short of Wilson's ideal, and even lies outside the historical realm of extensible programming. TsPyC does fulfil Wilson's goal of constructing a compiler which can be customised using what could be referred to as plug-ins; it does not, however, make any attempt to allow programmers to extend the language syntax. This is a deliberate choice, based on the idea that a programming language should make it at least as easy to write readable, maintainable code as possible. Redefinable syntax leaves programmers with too great an ability to construct unreadable programs.

1.2.2 Projects with Similar Aims

The aim of tsPyC was to provide programmers with the ease of use and expressibility that comes with the ability to introduce new language features, without sacrificing the efficiency associated with compiler programming languages. Numerous other projects have come at this problem from a slightly different angle. Such projects have noted that generally many interpreted programming languages already have good expressibility and ease of use. These projects have tackled the problem by attempting to reintroduce efficiency into such interpreted languages. Some of these projects are detailed in this section.

For an overview of methods which have been used to improve the performance of the Python language, including some listed below, see [7].

1.2.2.1 PyPy

The PyPy project [2] is centred around the primary goal of implementing a viable version of Python in Python itself.

> The PyPy project seeks to prove both on a research and a practical level the feasibility of constructing a virtual machine (VM) for a dynamic language in a dynamic language—in this case, Python. The aim is to translate (i.e. compile) the VM to arbitrary target environments, ranging in level from C/Posix to Smalltalk/Squeak via Java and CLI/.NET, while still being of reasonable efficiency within these environments. [13]

The PyPy virtual machine is written using a subset of Python (referred to as restricted Python, or RPython). The PyPy project has then written a tool-chain which may be used to translate the VM to some target environment. Commonly the VM is translated to C and then compiled. When tested against performance benchmarks, the compiled PyPy VM typically performs each iteration in between three and ten times the time taken by the standard distribution of Python, which is implemented in C [13].

The PyPy tool-chain can also be used to compile arbitrary programs from RPython to C or some other target language. In theory this allows programmers to use the features of the Python programming language, and to end up with machine code. In practice, there are drawbacks to this approach. One important obstacle faced by many newcomers to PyPy is that PyPy does not translate the whole range of the Python programming language, but only the restricted subset designated RPython by the PyPy project. RPython is not clearly defined or specified. In fact, the only detailed definition of what is

and is not allowed in RPython is the implementation. It is certainly an obstacle to programming for programmers to be unsure of whether certain code is or is not allowed in the programming language until they try to compile it.

One key concept in the PyPy project is the distinction between the code that is being compiled and code that will be executed at compile-time as part of the translation process. In PyPy, both these categories of code are written in Python (and some code may even fall into both categories), but the code that is to be compiled must be written using RPython. There is no such restriction on the code which is executed at compile-time as part of the translation tool-chain, which may be written using the full range of features available in the Python language.

1.2.2.2 Psyco

Psyco [11, 7] is a Python extension module which is designed to speed up the execution of Python code. Psyco is based on the concept of just-in-time (JIT) compiling, but might better be thought of as a just-in-time specialiser [12]. At run-time, it infers restrictions on variables from the values that a Python program manipulates. It then emits efficient machine code for the functions based on those restrictions. If data comes along later which does not match the inferred restrictions, Psyco can emit new machine code. The program is optimised at run-time for the data that it is currently handling.

Psyco has the advantage that existing Python code does not have to be modified in order to use it with Psyco. A programmer simply needs to include the Psyco module and the program will run with the performance benefits.

Running common Python code with Psyco typically results in a speed approximately four times that achieved by interpreting the Python code without Psyco. The performance gain varies depending on the code being executed. In situations where many repetitions and manipulations are performed on data of a fixed type, Psyco typically results in higher performance gains, up to 10 or 100 times that achieved without Psyco [11].

The seem to be several drawbacks of Psyco. Firstly, it is only implemented for Intel processors (although it does run independent of operating system). Secondly, it uses a lot of memory [11]. Another drawback is that complete native machine programs are not generated, so in order for customers to run software which uses Psyco, the customers must have Python installed on their computers.

1.2.2.3 Pyrex and Cython

Two interesting developments along a similar theme are the Pyrex project [9] and a fork of Pyrex known as Cython [14, 5]. Pyrex author Greg Ewing sums up Pyrex by saying "Pyrex is Python with C data types" [8]. Pyrex starts with Python code which is annotated to restrict the possible types of certain variables, and generates C code. In cases where data types are specified, C data types are used. For variables whose types are not specified, Pyrex will generate the needed C code to construct Python objects. Since extension modules are linked against the Python executables, almost all valid Python code is valid code in Pyrex.

Cython is an incomplete project which is based on Pyrex. Cython has the same essential goals as Pyrex, but provides a number of additional features [6]. Both the Pyrex and Cython projects build extension modules for Python, and require that Python be installed on the target system in order that software be executed.

1.2.2.4 Inlining C Code in Python

It is worth mentioning that there are a number of projects which have aims similar to those of this project in that they aim to improve the performance of high-level interpreted languages. In the case of Python, examples of software projects which allow some form of embedding of C code within Python include Cinpy [10], Weave [3, 7] and PyInline [1]. The aims of these projects differ significantly from those of the current project in that they aim to improve the performance of Python code without generating complete machine code for programs. They are mentioned here in order to give a more complete overview of the work that others are doing in the same area.

Chapter 2

Overview of Use

2.1 Process Overview

The tsPyC compiler takes two kinds of input: tsPyC source files and language extensions. Typical users would only need to concern themselves with writing source files. From a user's point of view, the tsPyC compiler is run on a source file, and either the compilation process succeeds and an executable file is generated, or the compilation process fails and relevant messages are displayed indicating the reason that compilation failed.

This process of compiling the source file takes place in three phases. The first phase involves parsing the input file and constructing an abstract syntax tree (AST). The syntax of tsPyC is fixed, and cannot be modified by language extensions. The second phase of the compiler is the tsPyC processor. The processor takes the AST and performs processing on it to construct a semantic tree. It is during the processor phase that language extensions are included. The final phase is to take the semantic tree and generate an executable output from that tree.

It is important to understand the distinction between the two different intermediate tree structures used within tsPyC. The first, the AST, is generated by the parser and is used to directly represent the structure of the input source file. The second is the semantic tree. This is used to represent the meaning of the source program. It is generated by the processor with help from language extension modules. It is this structure which is used to generate the output executable.

2.2 Example: Matrices

This section presents an example of the use of tsPyC. The purpose of this example is to give a broad understanding of the process used by tsPyC to compile source code. This section will not go into the details of the individual steps involved in the compilation process.

Listing 2.2 shows some example tsPyC source code which makes use of a language extension that adds matrices to tsPyC. As explained in Section 2.1, the first phase of the compilation process is the the parser phase. When provided with the example source code as input, the parser phase will output the AST depicted in Figure 2.2. The AST directly represents the structure of the input code. For instance, the matrix operation `A * C` is directly converted to a `binary_operation` node with two `IDENTIFIER` nodes as child nodes.

The AST is then provided as input to the processor phase, which converts it to a semantic tree.

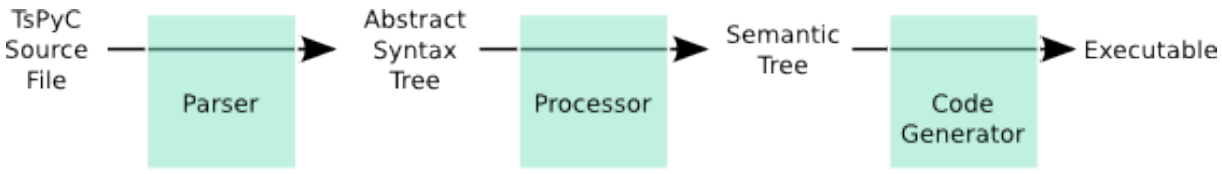

Figure 2.1: Phases of the TsPyC Compiler.

```
from matrices pymport matrix

begin program

__main__ := function()
    # Matrix literals.
    B := matrix
        1, 0
        0, 1

    C := matrix
        2
        3

    # Matrix variables.
    A: matrix(2, 2, int)
    X: matrix(2, 1, int)
    A = B

    # Element indexing.
    A[1,2] = 17

    # Matrix multiplication.
    X = A * C

    printf('%d %d\n', X[1,1], X[2,1])
```

Listing 2.1: TsPyC code which uses a matrices extension.

In this example, the AST in Figure 2.2 is converted to the semantic tree depicted in Figure 2.3. The semantic tree represents the intended meaning of the code in a form which can be used to simply generate output code. Notice that in the semantic tree, the matrix operation `A * C` is represented by a compound statement with assignments for each element of the resulting matrix.

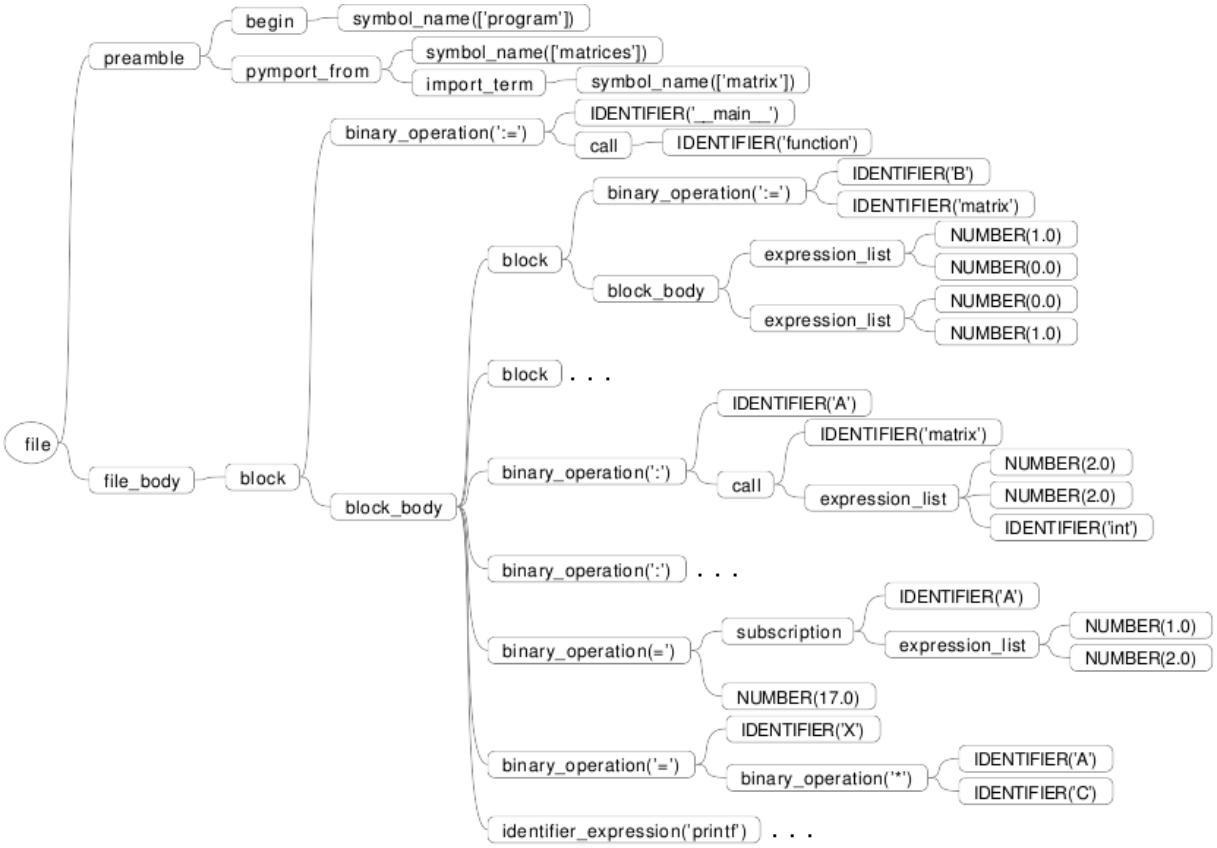

Figure 2.2: Partial AST structure for code in Listing 2.2.

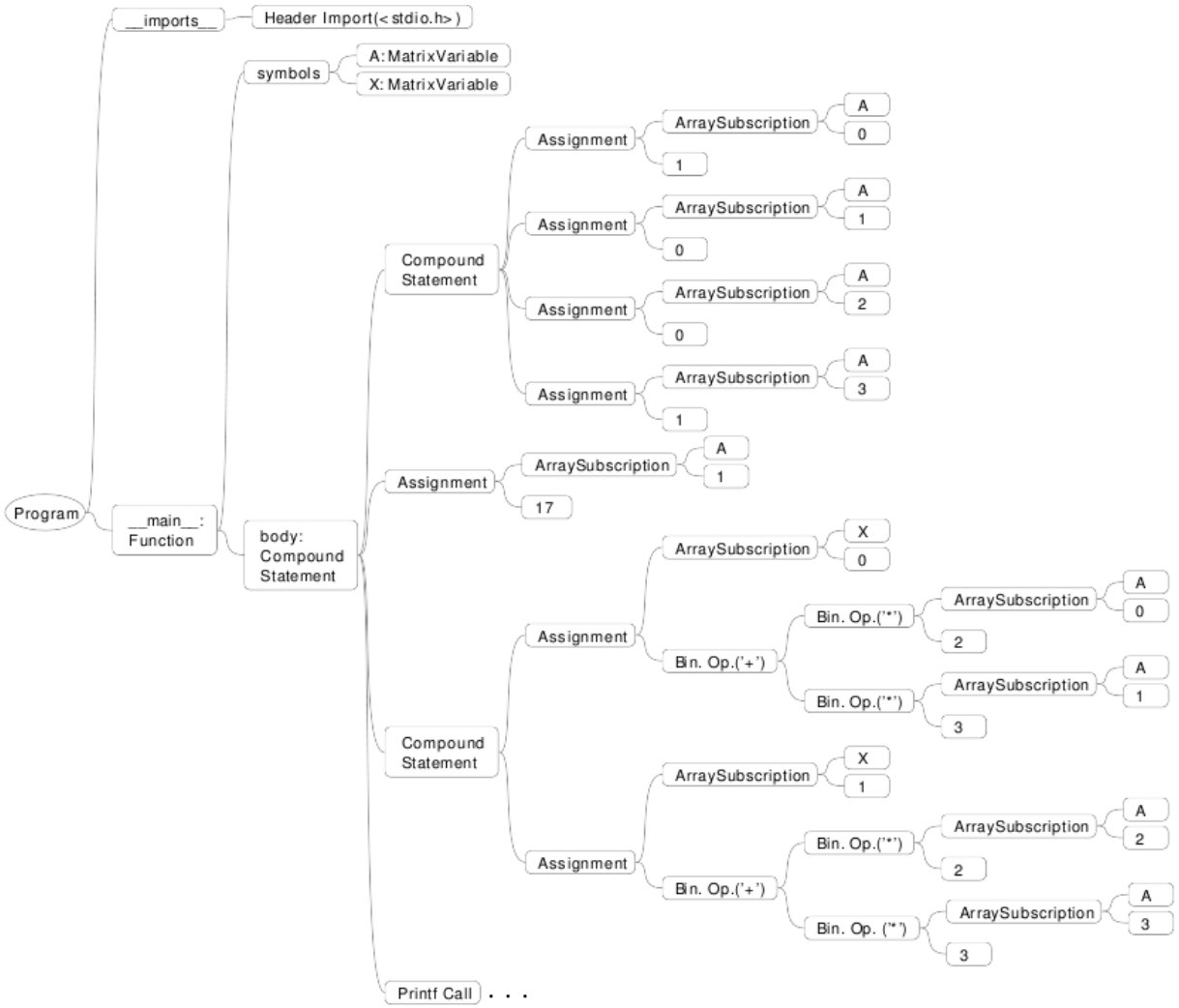

Figure 2.3: Partial semantic tree structure for AST in Figure 2.2.

Chapter 3

Language Syntax

3.1 Syntax Overview

This section gives an overview of the syntax of tsPyC. For the full language syntax definition, see Appendix A.

The syntax of tsPyC is built in to the language. That is, it can't be modified by extensions. That said, the syntax is very broad; it was written taking into account the fact that tsPyC exists to be extended. For instance, consider the matrices example in Listing 2.2. Even without the matrices extension, the following code is *syntactically* correct:

```
A := matrix
     1, 0
     0, 1
```

Without the inclusion of the matrices extension, which defines the `matrix` keyword, the code above has no *meaning*.

At the highest level, a tsPyC file is divided into a *preamble* and a *body* by a line starting with the keyword "begin". The preamble exists to tell tsPyC what different symbols should mean when interpreting the file body. The preamble contains directives which import objects defined in other tsPyC files or extensions into the symbol table which will be used by the processor.

Within the body of a tsPyC file, whitespace is used to determine high-level code structure. This is similar to the manner in which whitespace is used by Python—the indentation of a line of code determines the position of that code in the syntax tree. So in the code above, the lines expressing the contents of the matrix literal are considered to be part of a block which comes under the "`A := matrix`" line. They are part of this block by virtue of the fact that they are indented under that header line.

The building blocks for individual lines of tsPyC code within the body are identifiers, strings and numbers. These are joined almost exclusively by binary and unary operations and *suffix operations*. A suffix operation consists of an expression followed by a suffix of a particular form. The three suffix operations in tsPyC are called *subscription* (e.g. `expr1[expr2]`), *call* (e.g. `expr1(expr2)`) and *curly* (e.g. `expr1{expr2}`).

Table 3.1 shows the valid tsPyC operators in order of increasing precedence.

There is one more syntactic construct worthy of note in tsPyC. This is called the *keyword-guard* construct, and is formed by a single identifier (the keyword) followed by any expression (the guard). This construct is used in familiar language control structures such as `if guard`, or `while guard`. It may only occur at the beginning of a line, and is less tightly binding than any binary operation or subscription.

3.2 Syntax Design Decisions

The syntax of tsPyC was designed keeping in mind the fact that the language was intended for extension. This section documents the reasoning behind a number of design decisions relating to tsPyC's syntax.

Operation[a]	Example	Associativity
definition	`A := B`	non-associative
assignment	`A = B`	non-associative
outer mapping	`A -> B`	non-associative
list	`A, B, C`	flat[b]
inner mapping	`A : B`	non-associative
logical or	`A or B`	left-associative
logical and	`A and B`	left-associative
logical not	`not A`	right-associative
numerical comparisons	`A > B; A < B;` `A >= B; A <= B;` `A == B; A != B`	non-associative
bitwise or	`A | B`	left-associative
bitwise exclusive or	`A ^ B`	left-associative
bitwise and	`A & B`	left-associative
bitwise shift	`A << B; A >> B`	left-associative
addition, subtraction	`A + B; A - B`	left-associative
multiplication, integer division, division, modulo	`A * B; A // B;` `A / B; A % B`	left-associative
unary negative, bitwise negation	`-A; ~A`	right-associative
exponentiation	`A ** B`	right-associative
attribute access	`A . B`	left-associative
subscript operations	`A[B]; A(B); A{B}`	n/a
parentheses	`(A)`	n/a

[a] Note that these are only the meanings given to these operations by the base language—there is no reason why extensions need to restrict operations to these meanings.

[b] All expressions separated by commas will end up on the same level of the AST.

Table 3.1: Operator precedence in tsPyC, from least- to most-tightly-binding.

3.2.1 Fixed Syntax

Taking into account the fact that tsPyC is flexible and extensible, one of the most striking feature's of tsPyC's syntax is that it's fixed—at first glance it seems not to have the flexibility of the language itself. This was a deliberate decision made early in the planning of this project. The reasoning was that clear and readable code makes for maintainable code. Good programming languages should facilitate the development of maintainable software by encourage programmers to write such readable code. Allowing infinitely redefinable syntax was considered to detract from this objective by making it too easy to allow code to become unreadable.

Extensible syntax was a feature of many of the extensible programming of the 1960s and 70s. According to Standish, one of the reasons that extensible programming didn't take off was that a programmer had to be familiar with existing extensions in order to successfully write a new extension with any complexity [15]. TsPyC was designed to have modular extensions which should not need to know about one another in order to work. It is difficult to see how one could achieve such modular extensibility with a flexible language syntax.

3.2.2 Significant Whitespace

Whitespace at the beginning of a line is significant in tsPyC. It is used to determine the high-level syntactic structure of a program. This concept was borrowed from Python, but can also be seen in other languages such as Occam and Haskell. The rationale behind this decision is twofold. Firstly, the parser needed a device to allow it to gain some information about the structure of a source file without reference to any language semantics, which are flexible. Indentation was an obvious way to allow this.

Secondly, one of goals when designing tsPyC was to promote the writing of readable code. Using indentation is an intuitive way to delimit blocks with common meaning which results in readable code. In fact, most programmers try to use indentation to denote program structure in a readable way, even when using languages which use delimiters such as `{...}`. Readability is particularly important when extensions are involved, and regardless of the meaning which a particular extension gives to a particular block of code, using indentation is clear way of showing which code belongs together.

In Python code, for a line of code to be followed by an indented block, that line must end with a colon. TsPyC does not have this requirement, partly because a colon is already used as a binary operator in tsPyC, but also because a colon at the end of a line takes up space without seeming to significantly improve readability. The extra level of syntactic redundancy which such a colon provides was not considered necessary. The lack of colons may also serve as a reminder to programmers that it is not Python they are programming in.

3.2.3 Ubiquitous Expressions

TsPyC takes a different approach to some languages in that almost any expression, properly bracketed, may appear within another expression. For instance, the expression `a+(b=c)` is not a valid expression in Python, but is in tsPyC. Even tsPyC's "`:=`" operator, which is used to bind objects to names in the symbol table, is allowed to occur within other expressions. This is not because tsPyC wants to mimic C in its useful but often unreadable short-hand expressions. (In fact, in the tsPyC base language, the expression `a+(b=c)` will result in an error, but not a parser error.) It is for the simple reason of flexibility. An extension programmer may wish to assign any meaning to the symbols, and the extensions need a syntax with some room to move if they are to have the freedom to improve the expressibility of the language.

3.2.4 Operators and Precedence

TsPyC introduces a number of operators which are either uncommon, or would usually have special meaning. The definition operator, "`:=`", has a special significance to the tsPyC processor. When used as the top-level operation in a line, it binds objects to names within the current symbol table. For this reason it is given the lowest precedence of all the binary operators.

The comma operator is introduced in order to allow sequences to be expressed. In the base language this is primarily used for sequences of function parameters, but a sequence of comma-separated expressions

may be used anywhere that an expression is valid (see, for example, the matrix literal construct in Listing 2.2).

TsPyC introduces two mapping operators, ":" and "->". In Python, the colon is used to define mappings, as in `{key1: value1, key2: value2}`. The colon in tsPyC may be used in a similar role due to it being more tightly binding than the comma operator. It was considered useful to also have a mapping operator of lower precedence than the comma operator. This is what the "->" operator is for. An example of its use in the base language is in the header line of a function definition, such as:

```
fn := function(param1: t1, param2: t2 -> returnType)
    return param1
```

In many languages, the full stop (.) is used for attribute access. For instance, `obj.attr` would refer to the attribute named `attr` of the object named `obj`. The tsPyC base language follows this convention and uses the full stop for accessing members of data structures. The language syntax does not, however, restrict the use of the full stop to require that it be immediately followed by an identifier. If an extension intends that a full stop be followed by an identifier in a particular context, the extension must check to ensure that that is the case. The decision to treat the full stop operator the same as all other binary operators was made, as one might expect, for reasons of flexibility.

Beyond the operators mentioned above, operator precedence in tsPyC follows reasonably common conventions, and is based heavily on Python's operator precedence. One might argue that, because tsPyC is built to be extended, simply using standard conventions for operator precedence would not be enough. Perhaps operator precedence should be able to be customised to suit different contexts. While it is true that a given operator may, in some contexts, have a meaning other than the most common meaning, modifying the operator precedences equates to changing the language syntax. As noted previously, this is not a good idea. And since the operator precedences are to be fixed, it makes sense for them to be fixed in a manner which makes most sense and is most readable to the average user.

For completeness, it is worth noting that it technically *is* possible for an extension to, in a context specific to that extension, modify the precedence that an operator appears to have. But this can only be done at the processor phase of compilation, and can be achieved, for instance, by modifying an AST after it had been built by the parser.

3.2.5 Flexible Keywords

In order that customisations may be able to define new keywords, the keyword-guard construct was included in the language syntax to represent a keyword followed immediately by an expression. Such a construct is commonly used for control structures such as `if` and `while`. The reasoning behind introducing the keyword-guard construct was that extensions should be able to define control structures similar to `if` or `while` blocks. A truly flexible language should have as few special cases as possible.

The trouble with introducing such a construct into the language is that, if not handled correctly, it can lead to ambiguity. For instance, consider the following lines of code:

```
x-3             # Subtract 3 from x
fn(3)+1         # Call a function then add 1
return-3        # Return -3
return (3)+1    # Return 3+1
```

Although the meaning of these lines of code may seem clear to a human reader, they are only clear because most readers already have some concept of how they expect the `return` keyword to behave. Since the parser has no way of distinguishing the meanings of identifiers `x`, `fn` and `return`, it is clear that the parser must return the same AST structure for the expressions `x-3` and `return-3`. Similarly, `fn(3)+1` and `return(3)+1` must result in the same AST structure as one another.

When designing the syntax of tsPyC, the principles of flexibility and code readability were deemed to be important enough to still include the keyword-guard construct despite these problems. The problems were resolved in the following way. Firstly, an identifier followed by an expression is valid syntax for a keyword-guard construct *only* when it occurs at the start of a line. So if a line of source code said "`x = return - 3`", the parser would always consider `return - 3` to be a subtraction operation. Similarly, if a line of source code said "`x = return(3)`", the right hand side of the assignment will always be interpreted as a function call.

Secondly, the keyword-guard construct has the lowest precedence of operations which may occur on a line of source code. This means that each of the four lines of code shown above will be interpreted by the parser to be examples of the keyword-guard construct. This sometimes means that the parser will get the intended program structure wrong. Fortunately, because of the fact that a keyword-guard construct always occurs at the start of a line of code, when the tsPyC processor discovers that `x` or `fn` do not have any meaning when used as keywords, it only takes execution of a simple algorithm to modify the AST so as to obtain the intended structure. This modification is performed by the tsPyC processor based on whether or not an identifier is allowed to be used as the keyword of a keyword-guard expression, as determined by which interfaces it implements. Neither typical users nor extension writers need to concern themselves with this rearranging of the AST by the processor.

Chapter 4

Semantic Trees and Code Generation

The processor phase of the compilation process of tsPyC source code takes an AST and uses it to generate a semantic tree. This semantic tree represents the intended meaning of the program. This section describes the building blocks which make up such semantic trees, and discusses how semantic trees are used to generate compiler output.

4.1 Overview

Unlike the AST structure, the structure of the semantic tree in TsPyC is not fixed—it is possible to define new objects which may appear in a semantic tree. The semantic tree also does not follow the same rigid tree-like structure as the AST. Rather, the semantic tree is simply made up of any Python objects which fulfil certain well-defined interfaces.

The semantic tree represents the meaning of a program in a form that is closely tied to the output of the tsPyC compiler. TsPyC is designed to generate native machine code, but it does so by using C code as an intermediate. Ideally, this C code should not be visible to the user; the user should simply see tsPyC source code compiled to machine code. The reason that C is used as an intermediate step is as follows. Firstly, by using an existing compiler as a back-end, the tsPyC is free from having to reproduce the vast amounts of work done by others, and can focus on the extensibility of the front-end—what tsPyC was designed for. Using a C compiler as a back-end has the additional advantages that existing C compilers are retargetable to different CPUs, and already perform a fair amount of optimisation during their execution.

Due to the fact that C is used as an intermediate, the tsPyC code generator's job is to turn semantic trees into C code. Semantic trees were therefore designed to directly correlate to C code. The tsPyC base language provides a set of objects which can be used to build semantic trees. These objects are likely to be enough to build semantic trees for many programs, but to ensure maximum flexibility, it is possible to define new objects which can be used in semantic trees. To facilitate this, tsPyC has well defined interfaces which objects implement if they are to be a part of a semantic tree. These interfaces relate directly to the generation of C code. In short, for an object to be a part of the semantic tree, it must know how to generate its own C code. For a complete list of interfaces defined by tsPyC, see Appendix C.

4.2 Code Generation

The code generation phase of the compiler takes a semantic tree and converts it to C code which is then compiled. In order to do this, the code generator performs a traversal of the semantic tree, calling the code generation functions defined in the relevant interfaces for code generators (see Appendix C). For instance, to generate code for an object which acts as a statement in the semantic tree, the code generator phase would call the `generate_stmt()` function of the object. This is demonstrated in Listing 4.2 which provides the actual source code behind the "`CompoundStatement`" semantic tree node. Notice that it provides a `generate_stmt()` method which calls the `generate_stmt()` of each child node. Similar generation methods are provided for each of the different contexts of the semantic tree.

```
class CompoundStatement(object):
    '''Useful for when performing processing expects a single return value
    but a routine wishes to return zero or more statements.'''
    def __init__(self, statements):
        for statement in statements:
            if statement == ERROR:
                continue
            if not ismember(statement, STATEMENT_GENERATOR):
                raise CategoryError('%s is not a STATEMENT_GENERATOR' %
                    (statement,))

        self.statements = statements
        guarantee_membership(self, STATEMENT_GENERATOR)

    def generate_stmt(self, fd, indentation):
        for statement in self.statements:
            statement.generate_stmt(fd, indentation)
```

Listing 4.1: Implementation of the "`CompoundStatement`" semantic tree node, demonstrating code generation.

4.3 Building Blocks for Semantic Trees

TsPyC provides a set of building blocks for use constructing semantic trees, but also allows users to define their own building blocks. Table 4.2 lists the building blocks provided with the base language.

4.4 I Don't Want an Executable

It is entirely possible that a user may wish to use tsPyC to generate something other than executable code. While the base language is geared towards generating executable code through intermediate C code, it is possible to extend the language to generate other output. The code generation targets available for a particular tsPyC file, depend on the *tsPyC file type*, which is specified on the "begin" line of a tsPyC source file. For instance, the most common tsPyC file type is the `program` file type, which uses the default tsPyC processor and code generator. To indicate that this file type is being used, the source file contains the line `begin program`. A custom tsPyC file type could potentially make use of a different processor and code generator. An extension can provide a custom tsPyC file type by implementing the `FILE_TYPE` interface. For a complete list of interfaces defined by tsPyC, see Appendix C.

Name	Description
`AddressOf`	Take the address of an expression
`ArraySubscription`	Access an element of an array
`Assignment`	Assign to an expression
`BinaryOperation`	Binary operation
`CompoundStatement`	Multiple statements
`Declarations`	Contains variable declarations
`Dereference`	Dereference a pointer
`ExpressionStatement`	Use an expression as a statement
`ExternalFunctionCall`	Call a C function not defined in tsPyC code
`Function`	A function
`FunctionCall`	Call a function
`Goto`	Jump to a label
`HeaderImport`	Import a .h file
`IfStatement`	If / else control structure
`LabelPos`	Used to place a label
`LabelRef`	Used to refer to a label
`Literal`	A literal
`Loop`	While loop
`PrintfCall`	Call `printf()`—provided for convenience
`Program`	Envelope for entire .c file
`RawOutput`	Text to be output directly as C code
`ReturnStatement`	Return from a function
`ScanfCall`	Call `scanf()`—provided for convenience
`TransparentCoercion`	Treat a variable of one type as if it has another type
`TypeCast`	Cast a variable to a new type
`UnaryOperation`	Unary operation
`Variable`	A variable

Table 4.2: Semantic tree building blocks provided with the base tsPyC language

Chapter 5

Extensions and the Processor

The main use of tsPyC extensions comes during the processor phase, while the AST is being used to generate the semantic tree. This section outlines how the tsPyC processor behaves and how extensions may be defined.

5.1 Processor Overview

The tsPyC processor takes as input an AST and a symbol table. The AST directly represents the contents of the tsPyC source file. The symbol table contains definitions based on the imports specified in the preamble of the source file. Using the AST and symbol table, the processor follows well-defined rules to interact with the base language and extensions in order to generate a final semantic tree.

The procedure used by the processor is essentially to perform a depth-first traversal of the tree and process each node of the tree according to its context. The processor itself has no notion of the semantics that should be associated with any particular symbols, even symbols defined in the base language. From the processor's point of view, there is no difference between the base language and the extensions.

For every kind of node in the AST, the processor has a particular behaviour. As discussed in Section 3.1, the AST is constructed primarily as follows:

- Identifiers, numbers and strings make up leaf nodes;
- Leaf nodes are combined in expressions, made of binary, unary and suffix operations;
- Expressions are occasionally used in keyword-guard constructs;
- Expressions or keyword-guard constructs are used as lines;
- A line may be followed by a block of indented lines.

When it encounters a leaf node, the processor will simply resolve the identifier in the current symbol table, or construct a `Literal` object to contain the number or string. When it encounters any other node, it will perform a number of steps. Firstly, it will process the left-hand branch of the node. For instance, this may be the left-hand side of an addition expression, or may be the header line of an indented block. After the left hand side has been processed and resolved, the resulting object will be given the opportunity to determine the behaviour of the processor. An object may or may not provide customisation behaviour for any given context within the AST. Providing such customisation behaviour is done by implementing interfaces defined by tsPyC.

If the left-hand branch of a node does not provide customisation for a given context, and the object has a type, that type may provide customisations on the object's behalf. For example consider a variable of matrix type. The variable object may not provide customisations for when a variable is multiplied, but the matrix type may provide such customisations.

If neither the object on the left-hand branch of a node, nor its type, provides customisation for the given context, one of two things will happen. If the AST node is a unary or suffix operation, the processor will report an error to the user, indicating that the object in question is not valid in the current context. In all other contexts, the right-hand branch will be processed and given an opportunity to provide

customisations is situations in which the left-hand branch will not. For instance, in the expression `17 * M`, if `M` is a matrix, then the literal seventeen is not able to provide a customisation for that context. This is because literals are part of the base language, and do not know about matrices. But the `matrix` type is able to provide customisation behaviour for when a matrix is multiplied by a scalar, in this case seventeen, on the left-hand side.

5.2 Customisation Behaviour

A customisation routine is able to do two things. It may report an error which will be returned to the user on the error console. It may also return a section of semantic tree or other intermediate object, which should be used by the processor to represent the particular operation or structure corresponding to the context for which the customisation is being called.

A typical customisation routine would perform the following steps:

1. **Perform syntax checking on the AST.** This must be distinguished from the syntax of the language, which has already been used to construct the AST. This step is simply checking that, within the broad possibilities afforded by the generically-defined language syntax, the user has chosen the correct syntax constructs for this context. For instance, Listing 2.2 defined matrix types using expressions like `matrix(2, 2, int)`. AST syntax checking for such a situation might, for instance, check to make sure that the expression had exactly three comma-separated expressions in the brackets.

2. **Process any child nodes.** Customisations have the ability to call the tsPyC processor on AST nodes. This allows a user to make user of one customisation within the context of another customisation, without the developers of the two customisations having to know about one another's work.

3. **Perform static semantic checking.** For instance, in the case of the expression `matrix(2, 2, int)`, the customisation would check to ensure that the results of processing the first two parameters are integer literals, and that the result of processing the third parameter is a valid type.

4. **Construct the output.** This could be a semantic tree section, or could simply be an intermediate object which provides customisations for contexts further up the AST.

5.3 Example Extension Customisation

Section 2.2 introduced the example of a matrices extension, and partially depicted both the AST (Figure 2.2) and semantic tree (Figure 2.3) for that example. In this section, we will take a small part of the AST (a single matrix multiplication) and show the process used to transform it into the semantic tree. Listing 5.3 shows a few relevant sections of the example source code.

We will commence this example at the point in time when the processor is partway through traversing the AST. It has just reached the node corresponding to the statement `X = A * C`, but has not yet processed that node. The corresponding syntax tree is depicted in Figure 5.1. Note that at this point in time, the symbol `C` refers to a 2×1 matrix literal, `A` to a 2×2 matrix variable, and `X` to a 2×1 matrix variable.

To process the assignment node, the processor will perform the following steps:

1. The left branch of the node will be processed, which will resolve the symbol `X` to a 2×1 matrix variable.

2. The matrix variable object will be tested to see if it provides the customisations corresponding to an assignment. As it turns out, the matrix type does provide a customisation for when a matrix is assigned to. This customisation is called, and will:

   (a) Instruct the processor to process the right-hand branch of the assignment. The processor will:

      i. Process the left branch of the multiplication node, which will resolve the symbol `A` to a 2×2 matrix variable.

```
C := matrix
     2
     3

# Matrix variables.
A: matrix(2, 2, int)
X: matrix(2, 1, int)

# Matrix multiplication.
X = A * C
```

Listing 5.1: Sections of Listing 2.2 to be used to illustrate customisation.

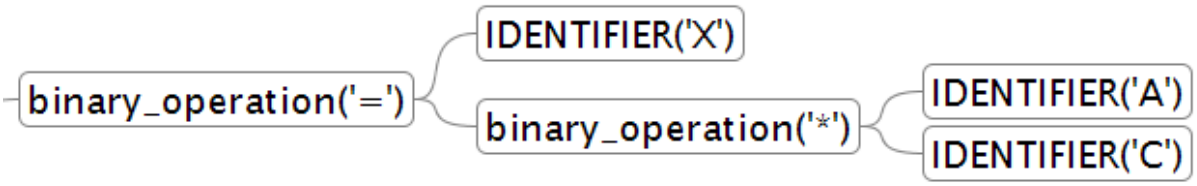

Figure 5.1: Syntax tree for matrix multiplication statement.

ii. The matrix variable object will be tested to see if it provides the customisations corresponding to a multiplication. The matrix type does provide these customisations, so the customisations will be called, and will:
   A. Tell the processor to process the right-hand branch of the multiplication. The processor will resolve the symbol `C` and return the corresponding 2×1 matrix literal.
   B. Test to see that the returned object is a valid matrix and that its dimensions are compatible with matrix `A`.
   C. Construct a matrix intermediate object representing the result of the matrix multiplication. This intermediate is represented in Figure 5.2. In order to construct the elements of this intermediate, the customisation calls the processor. The elements of the intermediate are constructed from `ArraySubscription` and `BinaryOperation` semantic tree nodes.
   D. Return this intermediate to the processor

iii. The processor will return this intermediate to the assignment customisation routine.

(b) The assignment customisation routine will then test to see that the returned object is a valid matrix and has the same dimensions as matrix `X`.

(c) A `CompoundStatement` semantic tree node will be constructed which assigns to each of the elements of `X`.

(d) This compound statement will be returned to the processor.

3. The returned value will be inserted into the semantic tree. The structure of this returned semantic tree node is depicted in Figure 5.3.

5.4 Design Decisions

When designing the tsPyC processor, there were a number of important considerations. In particular, it was considered critical for extensions to be self-contained and not interfere with one another—there

$$\begin{bmatrix} A[1,1] \times 2 + A[1,2] \times 3 \\ A[2,1] \times 2 + A[2,2] \times 3 \end{bmatrix}$$

Figure 5.2: The intermediate returned by the matrix multiplication customisation.

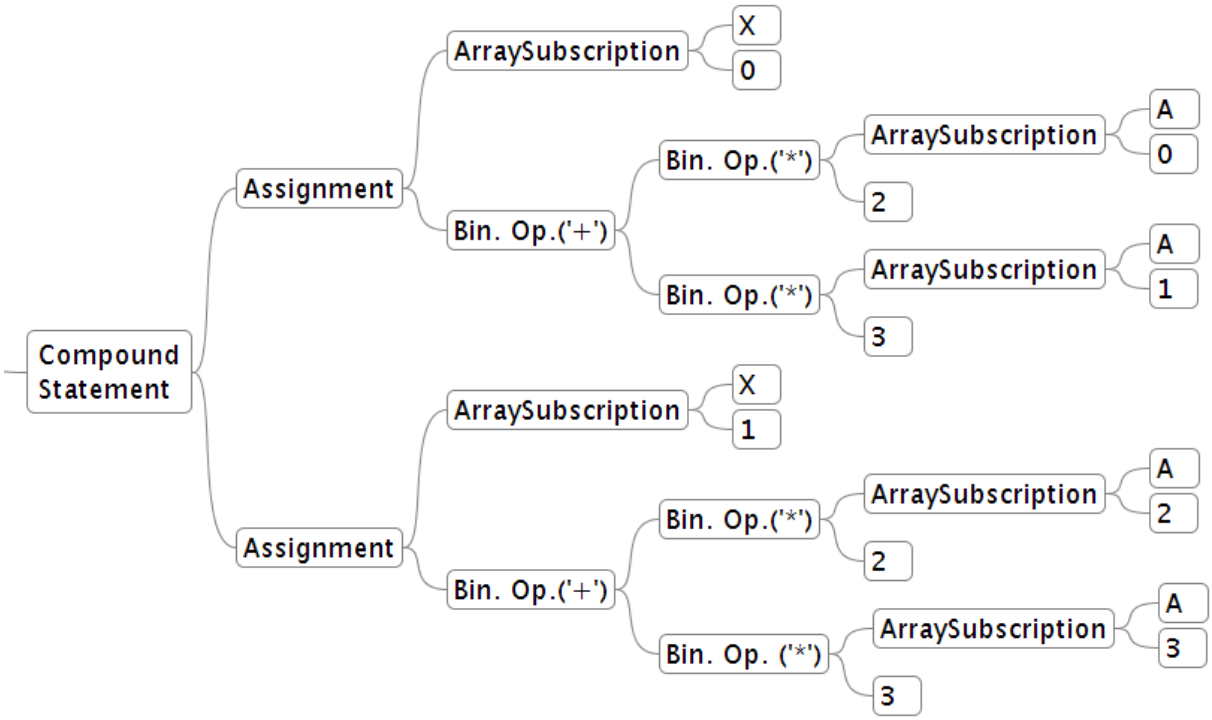

Figure 5.3: Semantic tree for matrix multiplication statement.

should be no situation in which there is any ambiguity as to which extension should take precedence. It was also important to give extensions the ability to use the language syntax to express concepts which are not part of the base language. Furthermore, the base language should not have any special privileges which are unavailable to extension modules.

5.4.1 Customisation Flexibility

The tsPyC processor provides more than simply a mechanism for operator overloading. Customisations can not only be defined for binary and unary operations, but for *any context in the syntax tree*. This was designed in this way so as to provide extensions with the ability to make use of the full scope of the tsPyC syntax. It was for this very reason that the tsPyC syntax was designed to be so broad.

When a customisation routine is called, it is usually passed as a parameter the AST branch to work with. For instance, the header line of an indented block is passed an AST node corresponding to the body of that block. Similarly, the (processed) left-hand side of a binary operation is passed an AST node corresponding to the right-hind node. This was done for reasons of flexibility. By passing in an AST branch to the customisation routine. The routine has the ability to perform syntax checking on the tree before continuing with its processing. This means that customisations can give special meaning to parts of the language syntax within particular regions of the code.

5.4.2 Error Handling

Extensions are provided with ability to add error messages to the compiler log. This allows for the construction of robust language extensions. In order to add an error to the compiler log, an extension can simply raise an exception of a specific type (`TsPyCError`). This exception is caught by the processor and converted into an error message. This mechanism was used because it seemed the most intuitive way for an extension writer to be able to signal that a user had made an error. All other exceptions are assumed to be the result of mistakes on the part of the extension programmer, and are propagated in Python's usual way, to allow the extension author to use their usual debugging techniques.

A second mechanism is also available for adding errors to the compiler log, by simply appending an object representing the error to a list of errors. This mechanism is provided primarily for instances in which the one customisation routine may wish to add multiple error messages to the log. In these situations, raising an exception will not suffice, because doing so would result in only the first error being reported to the user.

As an example of these two mechanisms of reporting compiler errors, consider Listings 5.4.2 and 5.4.2. Both demonstrate compiler errors; the former raises `TsPyCErrors` while the latter appends to the errors list.

It is worth noting that, while extensions perform error checking specific to the extension in question, the tsPyC processor also provides some level of error checking. The error checking provided by the processor amounts simply to checking whether the intermediate objects provided by extensions (or by the base language) are actually allowed to be used in the context in which they are used. That is, checking whether they implement the interfaces corresponding to the context in question.

5.4.3 Interface Definitions

The tsPyC processor needs to be able to tell whether a given intermediate object is allowed to be used in a particular context. This is done by testing to see whether the object implements a particular interface. TsPyC does this by making use of the `categories` module. TsPyC defines a number of interfaces (or categories). Extensions then need to guarantee that certain objects or classes of objects implement these interfaces (or technically, guarantee that these objects are members of the categories). Because Python is a dynamic, run-time language, objects are dynamically guaranteed to be members of categories at

```
class var(object):
    '''The var keyword should appear in the context of
        x := var(type)
    '''
    def __init__(self):
        guarantee_membership(self, SUFFIXABLE)

    def call(self, state, other):
        # First check for empty brackets.
        if other is None:
            raise TsPyCError('var() is invalid')

        # Process the body and expect a type.
        vartype = state.process(other)
        if vartype == ERROR:
            return ERROR

        if not ismember(vartype, TYPE):
            raise TsPyCError('expected: valid type', other.location)
        if ismember(vartype, ANCHORABLE) and vartype.anchor is None:
            raise TsPyCError('cannot create variable of unanchored type',
                other.location)

        # Create and return the variable object.
        return makevariable(vartype)

var = var()      # Singleton.
```

Listing 5.2: Base language definition of the "`var`" keyword, demonstrating error logging.

run-time. When the processor has an intermediate object, it can test to see whether the object does or does not implement a particular interface and behave accordingly.

From the point of view of an extension programmer, the key functions to know about are:

- `guarantee_membership(obj, category)`—guarantees that the given object satisfies the specification of the given category.
- `ismember(obj, category)`—tests whether an object is a member of the given category, i.e. whether `guarantee_membership(obj, category)` has been called previously.

Uses of each of these two functions can be seen in Listings 5.4.2 and 5.4.2.

The interfaces which tsPyC defines are generally specified as human-readable documentation which extension authors should take into account. When an extension makes a guarantee that a particular object fulfils a particular interface, the developer of that extension is guaranteeing to have read the documentation for that interface, and to have made that object fulfil all the specifications in the human-readable documentation. The full interface definitions may be found in Appendix C.

5.4.4 Symbol Scope Concerns

In order for a symbol to be able to be referred to from anywhere within its scope, the processor deals with the bodies of indented blocks slightly differently from other nodes: before processing individual lines in an indented block, the processor will collect all lines which contain only a single definition operation (i.e. `name := value`). These definition lines will be used to populate the symbol table for that scope. The tree nodes for the actual values are only processed lazily, to allow for things such as recursive function definitions.

The processor provides a facility for the construction of symbol tables for nested scopes. These symbol tables serve as barriers so that outer scopes cannot access symbols defined in inner scopes. Such a symbol table should typically be constructed for every indented block used by the base language or any extension.

```
class struct(object):
    '''Used to build struct types as follows:

    t := struct
        x: int
        y: int
    '''

    def __init__(self):
        guarantee_membership(self, BLOCK_HEADER)

    def processblock(self, state, blockbody):
        # Check the syntax of the struct block.
        assert ismember(blockbody, TREE_NODE)
        assert blockbody.kind == 'block_body'

        error = False
        lines = []
        for node in blockbody:
            matchresult = match(node,
                TreePattern('binary_operation', ':') << [
                    TreePattern('IDENTIFIER', name='id'),
                    TreePattern(name='type', edges=None)
                ])
            if matchresult is None:
                state.errors.append(CompilerError(node.location,
                        'invalid syntax--expected <name>: <type>'))
                error = True
            else:
                lines.append((matchresult['id'].value,
                              matchresult['type']))

        if error:
            return ERROR
        return StructType(lines)
struct = struct()   # Singleton.
```

Listing 5.3: Base language definition of the "`struct`" keyword, demonstrating error logging.

Due to the fact that the processor traverses the AST from top to bottom, failing to construct a symbol table for an indented block will sometimes result in unexpected results in terms of symbol visibility. This issue is a side-effect of the design of the tsPyC processor, and is not considered to be a serious drawback.

5.5 Base Language Design

The tsPyC base language was designed around the principle that the base language should not be given any special privileges not available to extension modules. Therefore, the base language was implemented in exactly the same way that extension modules are written, using exactly the same set of interfaces.

One key guiding principle here was that, if extensions are to be modular, they should not need to know about one another in order to work together. This same reasoning may be applied to the base language: the base language certainly does not know about the extensions, but the extensions should only need to know about the base language to the extent that they construct new base language structures. They should not assume that any given part of the AST corresponds to base language features. For instance, it is reasonable for a matrices extension to make use of the scalar addition and multiplication features of the base language to define matrix multiplication. But the matrices extension should not assume that the type of each element of a matrix must be one of the types defined in the base language.

The fact that the base language uses the same interfaces that any extension uses facilitates this extension modularity. It means that an extension can test whether a particular object implements the `Type` interface rather than testing whether the object is one of the base language types.

It is worth mentioning that it is possible for extensions to override or redefine base language concepts. The `environment` directive, which may be used in the preamble of a tsPyC source file, defines an extension module to load instead of the default environment, which is the tsPyC base language. By specifying an environment, none of the base language symbols (such as `program`, `function`, `int`, `if`, `return`) will be loaded. Rather, the symbols in the specified module will be loaded instead. Taken to the extreme, this customisation may be combined with the customisation mentioned in Section 4.4, resulting in what may look like a completely different language.

For a full description of the symbols defined in the base language and their associated semantics, see Appendix D.

Chapter 6

Discussion

6.1 Addressing the Aims

There were a number of aims in developing tsPyC. This section discusses how tsPyC addressed each of these aims.

6.1.1 Flexibility and Expressibility

TsPyC aimed to provide the flexibility for new language features to be expressed in source code with help from extension modules. The matrices example presented in Listing 2.2 demonstrates that this expressibility is possible in tsPyC.

TsPyC achieves this flexibility and expressibility through numerous design decisions. In particular, this flexibility was aided by the decision to have a broad syntax which allows more expressions to be syntactically valid than have meaning in the base language. This broad syntax, combined with the fact that the processor is designed to allow customisation in each different syntax tree context, provides a great deal of flexibility to the language.

The base language was designed to follow the same rules which extension modules must follow. This means that any language construct which appears in the base language may be mimicked and build upon by extensions modules. Extension modules also have the flexibility to make use of the compiler error console in the same way that the base language does.

Finally, the high-level customisation afforded by directives such as `begin` and `environment` provide developers with as much freedom as they could want to build upon the groundwork provided by tsPyC and its base language.

In evaluating the language, tsPyC certainly has the level of flexibility which was aimed for from the outset. The syntax seems to give the language the ability to simply express new concepts such as matrices or units. On this front, tsPyC should be considered a success.

6.1.2 Program Readability

One of the aims of tsPyC was for source code to be readable, even when such source code made use of language features defined in extensions.

The decision for tsPyC to have a fixed syntax did much to achieve this goal of program readability. Additionally, the use of indentation-based structuring of source code helps ensure that the meaning which tsPyC associates with source code is closely aligned (in terms of code structure) with the meaning a human reader would give it. TsPyC's operator precedences correspond closely to those of other languages, further aiding program readability.

Program readability is something that is difficult to test with limited resources. This is particularly true when it comes to testing potential readability of code which makes use of some extensions which someone may write in future. Therefore it is difficult to objectively say whether or not tsPyC has achieved this aim. What we can say is that the design decisions of tsPyC go some way towards encouraging readable source code in the language.

6.1.3 Extension Modularity

Another aim of tsPyC was for extensions to be modular—that is, to be self-contained packages which can be independently distributed and used. A number of design decisions contributed to this modularity.

TsPyC's fixed syntax avoids many problems associated with modularity. Had the syntax been extensible, it would have been difficult for tsPyC to achieve this aim of modularity in that modules would interfere with one another. Naturally, a fixed syntax alone does not guarantee extension modularity.

In order that extensions be able to work together, interfaces were set up for extensions to implement, and the base language was constructed using the same interfaces. Extensions which are written carefully and make correct use of these interfaces interfaces should be able to work together successfully.

It should be noted that extension authors need not be concerned about the possibility that a keyword defined in one extension will clash with a keyword defined in another—tsPyC provides the ability to give an alias to a symbol when it is imported.

TsPyC has achieved the goal of modularity in the sense that it is possible to write self-contained extension modules for the language, which do not rely on other extensions. In terms of extensions which do not know of one another working together, the success has been mixed. For instance, it is straightforward to make use of an extension such as matrices within a control structure (such as a repeat-until loop), defined by another extension. Doing so provides no hassles whatsoever. When it comes combining extensions with more similar behaviours, things become more difficult. For example, one might have the matrices extension discussed in this document, and might also have an extension which allows the definition of data types with units (e.g. metres). One might expect it to be possible to define matrices with units. Testing this concept with a simple implementation of both matrices and units revealed that while the two modules worked together when trying to define a matrix with elements of type "float in metres", the modules did not work so smoothly together when trying to define a type to be an entire matrix in metres. Investigation showed that the implementation of the units extension made certain assumptions about the underlying data types which did not hold in the case of matrices.

This illustrates that, while tsPyC makes it possible for extensions modules to work together, how well they will do so relies on how well the extension modules in question are designed and programmed. For perfect modularity, the modules themselves must be written perfectly.

6.1.4 Feature Robustness

A further aim of tsPyC was that features introduced in extensions should be able to be robust and to perform type checking and other static semantic checking. This aim is achieved through the ability of extensions to easily log error messages to the compiler console. The use of well-defined interfaces which extensions must implement also contributes to the overall robustness of the system.

On reflection, these features seem to have been enough for robust extensions to be developed. This has been apparent through the example extensions developed to date.

6.1.5 Machine Code Generation

TsPyC also aimed to be able to produce native machine code, and to be able to be retargeted to various CPUs. Through the simple decision to make use of C code as an intermediate step, which is then run through an existing compiler, this aim has been achieved.

6.2 Comparison with Other Approaches

Various other approaches have been taken in order to achieve some of the aims of this project. This section examines a few such approaches, and compares them to the tsPyC project.

6.2.1 Extensible Programming Approaches

The modern concept of Extensible Programming attempts to achieve the same flexibility to which tsPyC aspires. However, modern extensible programming languages seem focused on having extensible syntax [17]. This approach is completely different from tsPyC's; tsPyC avoids extensible syntax entirely. Languages with extensible syntax have advantages in terms of expressibility—there is little that cannot be

```
def main():
    B = Matrix([
        [1, 0],
        [0, 1]
    ])

    C = Matrix([
        [2],
        [3]
    ])

    A = B
    A[1,2] = 17
    X = A * C

    print X
```

Listing 6.1: Python code analogous to tsPyC code in Listing 2.2.

expressed when the very structure of a language is flexible. Although this may seem like a disadvantage on tsPyC's part, tsPyC's fixed syntax is broad enough to give the language great expressibility. Additionally, tsPyC has definite advantages over such extensible languages in the arenas of code readability and extension modularity. It is difficult for extensions to work together if they define completely different syntaxes from one another.

6.2.2 High-level Run-time Languages

One might argue that there is little need for a language like tsPyC when there are already high-level languages which provide as much customisability as you like at run time. For instance, Python already allows objects to be customised in many ways: how they behave when used in different binary and unary operations; how they behave when called as if they were functions; how they behave when an attempt is made to get or set the values of attributes; and so on.

For instance, Listing 6.2.2 shows Python code analogous to the tsPyC code in Listing 2.2. Instead of making use of tsPyC's compile-time flexibility, the listing makes use of Python's run-time flexibility. It assumes that a programmer has already written a "`Matrix`" class to use.

It is true that some languages, including Python, provide customisation to their programmers. Such languages do not generally provide quite the same level of customisation as tsPyC. For instance, you cannot normally define new language control structures analogous to "`if`" or "`while`".

The key difference between tsPyC and such languages is the time at which customisation occurs. TsPyC performs customisations at compile time; such languages perform customisations at run time. There are situations in which compile-time customisation has distinct advantages. For instance, consider an extension which checks for unit consistency in calculations. Deferring this unit checking to run time would result in a performance penalty; it would also mean that unit-related programming errors would not be detected as early.

TsPyC has the additional benefit that it compiles to native machine code rather than virtual machine code. This generally results in great performance improvement. It also means that tsPyC can be used to compile code for deployment environments such as embedded systems, where there is not enough memory to run a virtual machine.

6.2.3 Compiling Run-time Languages

Various approaches have been made to combine the flexibility of high-level run-time languages with the efficiency of languages which compile to native machine code. Such approaches clearly share some of the goals of tsPyC.

In particular, the PyPy project [13, 2], introduced in Section 1.2.2.1 includes a tool-chain designed to compile a subset of Python code to machine code. This could be used to attempt to achieve both

flexibility and efficiency by trying to compile some Python code (such as that in Listing 6.2.2 to machine code). As discussed previously, PyPy's biggest problem is that it doesn't clearly define which parts of the Python language are supported. TsPyC avoids this problem by starting at the ground and working up rather than starting with an existing language definition and trying to compile it.

6.3 Potential Drawbacks

Despite the fact that tsPyC achieves so many of its aims, there are a few drawbacks to tsPyC's approach. These are discussed in this section.

6.3.1 Compile-time Performance

In order to provide the greatest degree of flexibility to extension programmers, tsPyC extensions are written in Python. Python is a flexible, high-level, byte-compiled language. The disadvantage to this decision is the compile process is less efficient than if, for example, extensions were written in C. That is, the time taken to compile a tsPyC source file compared to that taken to compile in another language is equivalent to the time taken to run a Python program compared to the time taken to run a native executable. While the fact that a product has poor compile-time performance will have no effect on the end-users of that product, the compile-time performance does have a direct influence on the speed of development. Of course this is unlikely to be a problem for small programs.

6.3.2 Writing Extensions Carefully

As mentioned earlier, if developers are writing extensions for tsPyC, and want their extensions to interact nicely with those of other developers, they need to take great care not to make assumptions about the intermediate objects they are handling. It would be nicer if the language could somehow make it easier to write co-operative extensions, but it is unclear as to how exactly the language could be modified to do so. So it remains the case that extension programmers need to be careful about the assumptions they make.

Chapter 7

Conclusion

This project was a success in that it achieves the stated aim of developing a language with the flexibility to have language features added in the form of robust, modular extensions.

This project has achieved the outcome of designing and developing the new language tsPyC. This language takes a source file as input, and parses it according to a fixed but broad syntax. It then processes it with reference to the base language and any extension modules imported by the source file. This processing phase has great flexibility, and has the ability to be customised by various extension modules. The processor phase results in a semantic tree representing the intended behaviour of the source file, which is converted to C code and compiled. By making use of an existing C compiler, tsPyC achieves the aim of being retargetable for various CPUs.

Appendix A

Language Syntax

This section gives a overview of the syntax of tsPyC. Ambiguities are resolved according to a precedence table. The source code defining this parser (using the PLY framework) is provided in Appendix B.

The following syntax definition is formed by collecting all of the individual production definitions from the parser source code. Each production is written using the form of EBNF accepted as input by PLY.

```
file : preamble file_body
file : preamble_body
file : preamble error

preamble : preamble_body begin_line
preamble_body : preamble_body preamble_line NEWLINE
preamble_body :
preamble_body : preamble_body error NEWLINE
preamble_line : environment
              | import
              | import_from
              | pymport
              | pymport_from

environment : ENVIRONMENT symbol_name
pymport : PYMPORT import_terms
import : IMPORT import_terms
import_terms : import_term
import_terms : import_terms COMMA import_term
pymport_from : FROM symbol_name PYMPORT import_from_term
import_from : FROM symbol_name IMPORT import_from_term
import_from_term : MUL
import_from_term : import_terms
import_term : symbol_name
import_term : symbol_name AS symbol_name

begin_line : BEGIN symbol_name NEWLINE
begin_line : BEGIN error NEWLINE
symbol_name : IDENTIFIER
symbol_name : symbol_name DOT IDENTIFIER

file_body : block_body
file_body :
line : line_contents
     | expression_block
expression_block : line_contents block
block : INDENT block_body UNINDENT
block : INDENT error UNINDENT
block_body : block_body NEWLINE line
block_body : line
```

```
block_body : block_body NEWLINE error

line_contents : IDENTIFIER expression
line_contents : expression
expression : expression_list
expression_list : expression COMMA expression
expression : expression DEFINE expression
           | expression ASSIGN expression
           | expression R_ARROW expression
           | expression COLON expression
           | expression OR expression
           | expression AND expression
           | expression GREATER expression
           | expression LESS expression
           | expression GR_EQ expression
           | expression LS_EQ expression
           | expression EQUALS expression
           | expression NOT_EQ expression
           | expression BAR expression
           | expression CARET expression
           | expression AMP expression
           | expression SHL expression
           | expression SHR expression
           | expression PLUS expression
           | expression MINUS expression
           | expression MUL expression
           | expression INTDIV expression
           | expression DIV expression
           | expression MOD expression
           | expression POW expression
           | expression DOT expression
expression : NOT expression
           | MINUS expression      %prec UMINUS
           | TILDE expression
expression : primary

primary : primary suffix
primary : atom
suffix : subscription
       | call
       | curly
subscription : L_SQUARE expression R_SQUARE
subscription : L_SQUARE R_SQUARE
subscription : L_SQUARE error R_SQUARE
call : L_ROUND expression R_ROUND
call : L_ROUND R_ROUND
call : L_ROUND error R_ROUND
curly : L_CURLY expression R_CURLY
curly : L_CURLY R_CURLY
curly : L_CURLY error R_CURLY

atom : IDENTIFIER
atom : STRING
atom : NUMBER
atom : L_ROUND expression R_ROUND
atom : L_ROUND R_ROUND
atom : L_ROUND error R_ROUND
```

Appendix B

Complete Syntax Source

The following source file defines the syntax of the tsPyC language programmatically. For a full understanding of how precedences and error handling works in PLY, see the PLY documentation available at [4].

```
'''
parser.py
This file defines the tsPyC parser. The parser is defined using the PLY library.
'''
import os

import ply.yacc as yacc

from tspyc.parser.scanner import Scanner, tokens
import tspyc.parser
from tspyc.tree import TreeNode, Location
from tspyc.errors import ParseError

def YaccDefinition(start=None, tabmodule='parsetab', filename='<string>',
    outputdir=''):

    def MakeTreeNode(p, kind, value=None):
        if isinstance(p[1], TreeNode):
            return TreeNode(kind, value=value, location=p[1].location)

        elif isinstance(p[1], list):
            assert isinstance(p[1][0], TreeNode)
            return TreeNode(kind, value=value, location=p[1][0].location)

        # Get location from token.
        linenum = p.lineno(1)
        charindex = p.lexpos(1) - p.lexer.lnotab[linenum-1] + 1
        return TreeNode(kind, value=value,
            location=Location(filename, linenum, charindex))

    def p_file(p):
        '''file : preamble file_body'''
        p[0] = MakeTreeNode(p, 'file') << [p[1], p[2]]

    def p_file_error(p):
        '''file : preamble_body'''
        p[0] = TreeNode('error', location=Location(filename, 1))

    def p_file_error2(p):
        '''file : preamble error'''
        p[0] = MakeTreeNode(p, 'file') << [p[1], TreeNode('error')]
```

```
def p_preamble(p):
    '''preamble : preamble_body begin_line
    '''
    # First child node is a begin node.
    # Remaining child nodes are preamble_line nodes.

    # p[1] is already a list.
    p[0] = TreeNode('preamble', location=Location(filename, 1)) << [p[2]] +
        p[1]

def p_preamble_body(p):
    '''preamble_body : preamble_body preamble_line NEWLINE
    '''
    # Returns a list.
    p[0] = p[1]
    p[0].append(p[2])

def p_preamble_body_empty(p):
    '''preamble_body :'''
    p[0] = []

def p_preamble_body_error(p):
    '''preamble_body : preamble_body error NEWLINE'''
    p[0] = p[1]
    p[0].append(TreeNode('error'))

def p_preamble_line(p):
    '''preamble_line : environment
                     | import
                     | import_from
                     | pymport
                     | pymport_from
    '''
    p[0] = p[1]

def p_environment(p):
    '''environment : ENVIRONMENT symbol_name'''
    # Only child is a symbol_name node.
    p[0] = MakeTreeNode(p, 'environment') << p[2]

def p_pymport(p):
    '''pymport : PYMPORT import_terms'''
    # Children are import_term nodes.
    p[0] = MakeTreeNode(p, 'pymport') << p[2]

def p_import(p):
    '''import : IMPORT import_terms'''
    p[0] = MakeTreeNode(p, 'import') << p[2]

def p_import_terms_base(p):
    '''import_terms : import_term'''
    p[0] = [p[1]]

def p_import_terms(p):
    '''import_terms : import_terms COMMA import_term'''
    p[0] = p[1]
    p[0].append(p[3])

def p_pymport_from(p):
    '''pymport_from : FROM symbol_name PYMPORT import_from_term'''
```

```
    if p[4] == '*':
        # pymport_star's single child is a symbol_name
        p[0] = MakeTreeNode(p, 'pymport_star') << p[2]
    else:
        # pymport_from's first child is a symbol_name, all
        # remaining children are import_term or import_as nodes.
        p[0] = MakeTreeNode(p, 'pymport_from') << ([p[2]] + p[4])

def p_import_from(p):
    '''import_from : FROM symbol_name IMPORT import_from_term'''
    if p[4] == '*':
        p[0] = MakeTreeNode(p, 'import_star') << p[2]
    else:
        p[0] = MakeTreeNode(p, 'import_from') << ([p[2]] + p[4])

def p_import_from_term_star(p):
    '''import_from_term : MUL'''
    p[0] = '*'

def p_import_from_term(p):
    '''import_from_term : import_terms'''
    p[0] = p[1]

def p_import_term(p):
    '''import_term : symbol_name'''
    p[0] = MakeTreeNode(p, 'import_term') << p[1]

def p_import_term_as(p):
    '''import_term : symbol_name AS symbol_name'''
    p[0] = MakeTreeNode(p, 'import_as') << [p[1], p[3]]

def p_begin_line(p):
    '''begin_line : BEGIN symbol_name NEWLINE'''
    # Single child node is a 'symbol_name' node.
    p[0] = MakeTreeNode(p, 'begin') << p[2]

def p_begin_line_error(p):
    '''begin_line : BEGIN error NEWLINE'''
    p[0] = MakeTreeNode(p, 'begin') << TreeNode('error')

def p_symbol_name_base(p):
    '''symbol_name : IDENTIFIER'''
    # value is a list of strings
    p[0] = MakeTreeNode(p, 'symbol_name', [p[1]])

def p_symbol_name(p):
    '''symbol_name : symbol_name DOT IDENTIFIER'''
    p[0] = p[1]
    p[0].value.append(p[3])

def p_file_body(p):
    '''file_body : block_body'''
    p[0] = MakeTreeNode(p, 'file_body') << p[1]

def p_file_body_base(p):
    '''file_body : '''
    p[0] = MakeTreeNode(p, 'file_body')

def p_line(p):
    '''line : line_contents
            | expression_block
```

```
    '''
    p[0] = p[1]

def p_expression_block(p):
    '''expression_block : line_contents block'''
    p[0] = MakeTreeNode(p, 'block') << [p[1], TreeNode('block_body') <<
        p[2]]

def p_block(p):
    '''block : INDENT block_body UNINDENT'''
    p[0] = p[2]

def p_block_error(p):
    '''block : INDENT error UNINDENT'''
    p[0] = [TreeNode('error')]

def p_block_body(p):
    '''block_body : block_body NEWLINE line'''
    # returns non-empty list of line nodes
    # may contain error nodes
    p[0] = p[1]
    p[0].append(p[3])

def p_block_body_base(p):
    '''block_body : line'''
    p[0] = [p[1]]

def p_block_body_error(p):
    '''block_body : block_body NEWLINE error'''
    p[0] = p[1]
    p[0].append(TreeNode('error'))

def p_ident_expression(p):
    '''line_contents : IDENTIFIER expression'''
    # Keyword-guard construct
    # For use in statements such as 'if foo'
    p[0] = MakeTreeNode(p, 'identifier_expression', p[1]) << p[2]

def p_line_contents_base(p):
    '''line_contents : expression'''
    p[0] = p[1]

def p_expression_base(p):
    '''expression : expression_list'''
    p[0] = p[1]

def p_expression_list_base(p):
    '''expression_list : expression COMMA expression'''
    if p[1].kind == 'expression_list':
        p[0] = p[1]
        p[1].edges.append(p[3])
    else:
        p[0] = MakeTreeNode(p, 'expression_list') << [p[1], p[3]]

def p_bin_expression(p):
    '''expression : expression DEFINE expression
                  | expression ASSIGN expression
                  | expression R_ARROW expression
                  | expression COLON expression
                  | expression OR expression
                  | expression AND expression
```

```
                    | expression GREATER expression
                    | expression LESS expression
                    | expression GR_EQ expression
                    | expression LS_EQ expression
                    | expression EQUALS expression
                    | expression NOT_EQ expression
                    | expression BAR expression
                    | expression CARET expression
                    | expression AMP expression
                    | expression SHL expression
                    | expression SHR expression
                    | expression PLUS expression
                    | expression MINUS expression
                    | expression MUL expression
                    | expression INTDIV expression
                    | expression DIV expression
                    | expression MOD expression
                    | expression POW expression
                    | expression DOT expression
    '''
    p[0] = MakeTreeNode(p, 'binary_operation', p[2]) << [p[1], p[3]]

def p_un_expression(p):
    '''expression : NOT expression
                  | MINUS expression      %prec UMINUS
                  | TILDE expression
    '''
    p[0] = MakeTreeNode(p, 'unary_operation', p[1]) << p[2]

def p_prim_expression(p):
    '''expression : primary'''
    p[0] = p[1]

def p_primary(p):
    '''primary : primary suffix'''
    p[0] = p[2]
    p[0].edges.insert(0, p[1])

def p_primary_base(p):
    '''primary : atom'''
    p[0] = p[1]

def p_suffix(p):
    '''suffix : subscription
              | call
              | curly
    '''
    p[0] = p[1]

def p_subscription(p):
    '''subscription : L_SQUARE expression R_SQUARE'''
    p[0] = MakeTreeNode(p, 'subscription') << p[2]

def p_subscription_empty(p):
    '''subscription : L_SQUARE R_SQUARE'''
    p[0] = MakeTreeNode(p, 'subscription')

def p_subscription_error(p):
    '''subscription : L_SQUARE error R_SQUARE'''
    p[0] = MakeTreeNode(p, 'subscription') << TreeNode('error')
```

```
    def p_call(p):
        '''call : L_ROUND expression R_ROUND'''
        p[0] = MakeTreeNode(p, 'call') << p[2]

    def p_call_empty(p):
        '''call : L_ROUND R_ROUND'''
        p[0] = MakeTreeNode(p, 'call')

    def p_call_error(p):
        '''call : L_ROUND error R_ROUND'''
        p[0] = MakeTreeNode(p, 'call') << TreeNode('error')

    def p_curly(p):
        '''curly : L_CURLY expression R_CURLY'''
        p[0] = MakeTreeNode(p, 'curly') << p[2]

    def p_curly_empty(p):
        '''curly : L_CURLY R_CURLY'''
        p[0] = MakeTreeNode(p, 'curly')

    def p_curly_error(p):
        '''curly : L_CURLY error R_CURLY'''
        p[0] = MakeTreeNode(p, 'curly') << TreeNode('error')

    def p_atom(p):
        '''atom : IDENTIFIER'''
        p[0] = MakeTreeNode(p, 'IDENTIFIER', p[1])
    def p_atom_str(p):
        '''atom : STRING'''
        # Interpret any escape codes in the string.
        try:
            str_val = p[1].decode('string-escape')
        except ValueError, E:
            linenum = p.lineno(1)
            charindex = p.lexpos(1) - p.lexer.lnotab[linenum-1] + 1
            loc = Location(filename, linenum, charindex)
            errors.append(ParseError(loc, 'Syntax error: %s' % E.args[0]))
            p[0] = MakeTreeNode(p, 'error')
        else:
            p[0] = MakeTreeNode(p, 'STRING', str_val)

    def p_atom_num(p):
        '''atom : NUMBER'''
        p[0] = MakeTreeNode(p, 'NUMBER', p[1])

    def p_grouping(p):
        '''atom : L_ROUND expression R_ROUND'''
        p[0] = MakeTreeNode(p, 'parentheses') << p[2]

    def p_grouping_empty(p):
        '''atom : L_ROUND R_ROUND'''
        # The main reason for this production is so that a line like:
        #   if () + 1
        # can be parsed to a valid tree structure, then interpretted later
        # based on whether "if" is a keyword or a function.
        p[0] = MakeTreeNode(p, 'parentheses')

    def p_grouping_error(p):
        '''atom : L_ROUND error R_ROUND'''
        p[0] = MakeTreeNode(p, 'error')
```

```
    precedence = (
        ('nonassoc', 'IDENTIFIER'),   # e.g. "if foo"
        ('nonassoc', 'DEFINE'),
        ('nonassoc', 'ASSIGN'),
        ('nonassoc', 'R_ARROW'),
        ('left', 'COMMA'),
        ('nonassoc', 'COLON'),
        ('left', 'OR'),
        ('left', 'AND'),
        ('right', 'NOT'),
        ('nonassoc', 'GREATER', 'LESS', 'GR_EQ', 'LS_EQ', 'EQUALS', 'NOT_EQ'),
        ('left', 'BAR'),
        ('left', 'CARET'),
        ('left', 'AMP'),
        ('left', 'SHL', 'SHR'),
        ('left', 'PLUS', 'MINUS'),
        ('left', 'MUL', 'INTDIV', 'DIV', 'MOD'),
        ('right', 'UMINUS', 'TILDE'),
        ('right', 'POW'),
        ('left', 'DOT'),
        ('nonassoc', 'L_ROUND'),
    )

    errors = []
    def p_error(p):
        if p is None:
            errors.append(ParseError(Location(filename, -1), 'Unexpected end of
                file'))
        else:
            linenum = p.lineno
            charindex = p.lexpos - p.lexer.lnotab[linenum-1] + 1
            loc = Location(filename, linenum, charindex)
            if p.type == 'SYNTAX_ERROR':
                errors.append(ParseError(loc, 'Syntax error: %s' % p.value))
            else:
                errors.append(ParseError(loc, 'Syntax error: %s token not
                    expected here' % p.type))

    parser = yacc.yacc(start=start, tabmodule=tabmodule, outputdir=outputdir)
    return parser, errors

class Parse(object):
    def __init__(self, text, production=None, lexer=None, filename='<string>'):
        if lexer is None:
            lexer = Scanner()

        if production is None:
            tabmodule = 'tspyc.parser.parsetab.parsetab'
        else:
            tabmodule = 'tspyc.parser.parsetab.parsetab_%s' % production

        parser, self.errors = YaccDefinition(start=production,
            tabmodule=tabmodule, filename=filename,
                outputdir=os.path.join(tspyc.parser.__path__[0], 'parsetab'))

        lexer.input(text)
        self.tree = parser.parse(lexer = lexer)
```

Appendix C

TsPyC Interface Definitions

The following partial source file listings describe all the interfaces made available in tsPyC.

```
'''
base.py
Module: tspyc.base
This file contains a number of simple definitions.
'''
from categories import *

# A valid FILE_TYPE must provide the following methods:
# * get_valid_targets() - must return a mapping of target name to target
    description.
# * build(obj, ast, symbols, errors) - when given a TSPYC_PROTO_MODULE, an
#       AST and a mapping of symbol name to symbol, must fill out the details
#       of the object so that it is a valid TSPYC_MODULE.
# * generate(obj, target, **params) - when given an object that resulted from
#       the build() method and a valid target listed in getValidTargets(), must
#       return generated output. This method may be called with a target of
#       None to perform generation for the default target. If the FILE_TYPE has
#       no default target, it should check for a target of None and raise an
#       appropriate error.
FILE_TYPE = Category()

# A TSPYC_PROTO_MODULE is passed to the FILE_TYPE.build() routine. It must have
# the following attributes:
#   .modulename - will either be an empty string or a string representing the
#       name of the module which this TSPYC_PROTO_MODULE will represent.
#   .filename - will either be an empty string or a string representing the
#       name of the file from which the module was loaded.
TSPYC_PROTO_MODULE = Category()

# A TSPYC_MODULE must have the following attributes:
#   .symbols - must be a mapping which exposes the symbols of the module
#       which can be imported by other modules.
#   .file_type - the FILE_TYPE object which created this TSPYC_MODULE. This will
#       automatically be set on a TSPYC_MODULE before it's passed to
#       file_type.build().
TSPYC_MODULE = Category(TSPYC_PROTO_MODULE)

'''
base.py
Module: tspyc.program.base
This file contains numerous basic definitions used by the tsPyC processor.
'''

from categories import *
```

```
##################
# Categories
##################

# A BLOCK_HEADER is any object which is allowed to appear as the header line
# of a block. For instance, consider the block:
#       while x > 3:
#           output(x)
# In this block, the object that results when the tree for "while x > 3" is
# processed must be a BLOCK_HEADER.
# A BLOCK_HEADER must have the following methods:
#   .processblock(state, block_body_node) - must return the
#       tree node that results from the construction of a block with the given
#       block body node. This method may raise TsPyCError(message) to indicate
#       that there was an error in the tsPyC code (although a similar effect
#       may be achieved by appending a COMPILER_ERROR to state.errors.
#       The state parameter will have attributes symbols, globals and errors.
BLOCK_HEADER = Category()

# A FOLLOWING_BLOCK is an object whose appearance in a block forms some kind
# of combination with the previous object in the block. For instance, consider
# the block:
#       if x % 2 == 1
#           x = x / 2
#       else
#           x = 3 * x + 1
# In this block, the first two lines are first processed into an IfStatement,
# then the last two lines are processed into an ElseBlock. The ElseBlock is a
# FOLLOWING_BLOCK because during processing the block is combined with the
# IfStatement.
# A FOLLOWING_BLOCK must have the following methods:
#   .processfollower(state, previous) - must return the object that results
#       from the occurrence of the current block following on from the object
#       previous. This method may raise TsPyCError(message) to indicate that
#       there was an error in the tsPyC code.
#       The state parameter will have attributes symbols, globals and errors.
FOLLOWING_BLOCK = Category()

# KEYWORD is like BLOCK_HEADER, but:
#  * It's used for a keyword, such as 'while' in the expression:
#       while x > 3
#  * The method which is called is .processkeyword()
KEYWORD = Category()

# If an object obj is an OPERABLE and the object appears in a syntax tree in
# the contex of (obj * node) where * is some binary operation,
# obj.method(state, node) will be called where method is a
# method name corresponding to the operation * (see program._binary_routines for
# the complete list). If this routine returns NotImplemented or the routine does
# not exist, the node will be treated as if no customisation is defined. The
# customisation may also return an instance of ChildProcessed (see comments in
# proc_binary_operation() for more info. If the customisation does not return
# NotImplemented or ChildProcessed, the return value of the routine will be
# used as the result of processing.
# Similarly, if (* obj) occurs where * is some unary operator, a customisation
# method will be called (see program._unary_routines, proc_unary_operation()).
# If (o1 * obj) occurs and o1 is not an OPERABLE or does not define the
# appropriate customisation, a customisation of obj may be called.
OPERABLE = Category()
```

```
# If an object is TYPED, it has some type. It must implement:
#   .type - must be a valid TYPE object. This attribute must be able to be
#       written to with another valid TYPE object.
TYPED = Category()

# For a TYPE object t:
#   * Methods with "inst_" names are expected to be customisations and are
#       expected to comply with the various method signatures. (Generally
#       inst_*(state, obj, other) for binary operations and
#       inst_*(state, obj) for unary operations.)
#   * Beyond binary and unary operations, the allowable inst_* methods are:
#       - inst_call(state, obj, other)
#       - inst_subscription(state, obj, other)
#       - inst_curly(state, obj, other)
#       - inst_varassign(state, obj, other) - special routine which, if
#           defined, overrides default behaviour when a variable of this type
#           is assigned to.
#   * t.coerce(x) must be defined, it must accept a single
#       object x as its parameter and if the object
#       x can be represented as a member of the type t this routine should
#       return an object representing the required coercion from object x to
#       type t (this may return the unmodified object x). If the object x cannot
#       be represented as a member of type t, should raise CoercionError.
#       Note that x may or may not be TYPED. Note also that this routine needs
#       to explicitly test for the case when x is typed and x.type == t
#       because there is no automatic test for this. The returned object must
#       be able to be used in a position where an object of type t is required
#       but does not have to have type t itself.
#   * t.rcoerce(x, s) may be defined, and if so it must accept as parameters an
#       object x of type t and a type s. If the object x can be represented as
#       type s, the coerced object should be returned. Otherwise CoercionError
#       should be raised.
#   * t.storagetype must be a valid STORAGE_TYPE object, indicating what type
#       should be used when this code is output as C code.
#   * t.variable_class may be defined, and if so it should be a valid
#       VARIABLE_CLASS. When var(t) occurs in tsPyC code, t.variable_class(t)
#       will be called to construct the variable. If no variable_class is
#       specified, the default Variable class (defined in
#       tspyc.program.variables) will be used. See also the makevariable()
#       function.
#   * For the purposes of readable error messages, __str__ should be defined
#       to return a meaningful type name.
TYPE = Category()

# A STORAGE_TYPE is a TYPE object which represents a C type and knows how to
# generate the corresponding C code.
# A STORAGE_TYPE must define:
#   .generatetype([name]) - must return a string representing this type given
#       that name is a string representing the type name. Note that name may
#       be an empty string, or may be a combination of a name and some other
#       symbols.
#   .storagetype - must be equal to the object itself.
STORAGE_TYPE = Category(TYPE)

# An UNPOINTABLE_TYPE is a valid TYPE to which pointers are not allowed. An
# error will be reported on an attempt to create a pointer type for an
# UNPOINTABLE_TYPE, or a type whose STORAGE_TYPE is an UNPOINTABLE_TYPE.
UNPOINTABLE_TYPE = Category(TYPE)

# For a CONDITION_TYPE t:
#   * t must be a valid TYPE
```

```
#   * nodes which have this type are allowed to appear as boolean conditions.
CONDITION_TYPE = Category(TYPE)

# A VARIABLE is an object designed to represent a tsPyC variable. While there
# are no hard and fast rules on how it should behave, you should keep in mind
# how users expect variables to behave when constructing VARIABLEs.
VARIABLE = Category()

# A VARIABLE_CLASS is an object which is used to construct a variable. It
# must:
#   * be callable
#   * accept one parameter which is a valid TYPE
#   * return a VARIABLE
# Note that a good way to create a VARIABLE_CLASS is to subclass the Variable
# class defined in tspyc.program.variables.
VARIABLE_CLASS = Category()

# A SUFFIXABLE object may define the following methods:
#   .subscription(state, node) - for obj[node]
#   .call(state, node) - for obj(node)
#   .curly(state, node) - for obj{node}
# In a particular case, if the appropriate method is defined, it will be called
# to perform customisations. The customisation method should return an object
# to be inserted into the syntax tree, or return NotImplemented, which is
# treated in the same way as if the customisation was not defined (usually this
# means that an error is reported). Note that if an object is not SUFFIXABLE,
# the customisation routine is not even called.
SUFFIXABLE = Category()

# A COMMAND object is an object which can be used as a single-line command.
# Examples of such commands are the break, continue and pass keywords.
# A COMMAND must define the following method:
#   .processcommand(state)
COMMAND = Category()

# An UNPROCESSED_SYMBOL is a symbol which has not been processed at the time
# that it is entered into the ProgramSymbolTable. At the time that it is read
# from the symbol table, its .process(name) method will be called and the result
# will be cached in the symbol table and returned from the get.
UNPROCESSED_SYMBOL = Category()

# An ANCHORABLE is an object which can be achored to a particular scope. This
# is usually used for definition lines which need to generate some code. For
# instance, x := var(int) needs to generate the code "int x;". This code should
# only be generated in one position, and any other places where the object is
# encountered should generate references to the code.
# The anchoring process works in two steps:
#   1. The first time that the ANCHORABLE object is used in a tsPyC scope, an
#       anchor is created for the object by the scope when the scope is
#       processed. The object's .set_anchor() method is called.
#   2. At some point the anchor will be bound to the C scope in which the
#       declaration code will actually be generated. This does not effect the
#       object - only the anchor's .name property is changed.
# An ANCHORABLE must have the following attribute:
#   .set_anchor(anchor, errors) - this will be called once to anchor the
#       object, with a valid ANCHOR as the first parameter and a list of
#       COMPILER_ERRORS which may be appended to.
#   .anchor - this should be None before the object has been anchored, and a
#       valid ANCHOR object otherwise. This attribute may be read-only.
ANCHORABLE = Category()
```

```
# An ANCHOR records information about the scope to which an ANCHORABLE is
# anchored. See the comment on the ANCHORABLE category for more information
# about the anchoring process.
# An ANCHOR must have the following attributes:
#   .name - before the anchor is bound, this is a name which may represent the
#       name which was given to the object in the tsPyC code. After the anchor
#       is bound, this attribute will be the C identifier name with which this
#       anchor is associated.
#   .module - is the tspyc.tree.Module to which this symbol was anchored.
#   .state - a tspyc.program.ProcessorState object indicating the namespaces
#       to which this anchor is bound.
#   .location - the location of the line which causes this anchor to be created.
ANCHOR = Category()

# A NAME_ONLY_ANCHORABLE object is an ANCHORABLE object which is anchored purely
# so that it can receive a C identifier as a name. Being anchored does not
# generate any additional c code. This category is used by labels so that they
# can obtain names which do not collide with other symbols, but labels do not
# generate declaration code in the same way that variables do.
NAME_ONLY_ANCHORABLE = Category(ANCHORABLE)

# A PROTOTYPE_GENERATOR is an object which is allowed to have a prototype
# generated for it at the top level of a C module.  It must have:
#   .generate_prototype(fd, module, level) - must write to the given file-like
#       object any prototype code which this object wishes to generate. This
#       will be called for all TOP_LEVEL_GENERATOR objects which are referenced
#       in the module, even if they are anchored to a different module. The
#       module parameter is the tree.Module in which the reference to the
#       definition is being made. Note that if prototype_levels is an empty
#       sequence, this function is allowed to not be defined.
#   .prototype_levels - must be a sequence of numbers indicating the positions
#       of different prototypes that this object generates. Prototypes of lower
#       level are generated first in the output file. .generate_prototype() will
#       be called exactly once for each level in .prototype_levels.
#       Levels currently used by tsPyC are:
#           20 - header import
#           50 - forward struct definitions
#           100 - type definitions
#           200 - function prototypes
#           300 - external global variables
PROTOTYPE_GENERATOR = Category()

# A TOP_LEVEL_GENERATOR is an object which is allowed to appear at the top
# level of a C module. It must have:
#   .generate(fd, module, level) - must write to the given file-like object any
#       code which this object wishes to generate. This will be called for all
#       TOP_LEVEL_GENERATOR which are in the global namespace. It is up to
#       the object to check if it is anchored in this module. The module
#       parameter is the tree.Module in which the object is referred to. Note
#       that if code_levels is an empty sequence, this function is allowed to
#       not be defined.
#   .code_levels - must be a sequence of numbers indicating the positions of
#       different bits of code that this object generates. Output code will
#       always appear after all prototype code, and code with lower level will
#       be generated first in the output file.
#       Levels currently used by tsPyC are:
#           100 - global variables
#           200 - function code
# The .generate_prototype() method will be called for all top-level generators
# before any .generate() methods are called.
TOP_LEVEL_GENERATOR = Category(PROTOTYPE_GENERATOR)
```

```
# A STATEMENT_GENERATOR is an object which is allowed to appear as a statement
    in
# a C module.
# A STATEMENT_GENERATOR must define:
#   .generate_stmt(fd, indentation) - write to the given file-like object any
#       generated code. Indentation will be a non-negative integer suggesting
#       the indentation level which should be used.
#       This routine may assume that the current location of fd is one at which
#       zero or more statements are expected, and must leave fd in a location at
#       which one or more statements are expected.
STATEMENT_GENERATOR = Category()

# A DECLARATION_GENERATOR is an object which, when anchored, generates a
# declaration (for example, a variable).
# A DECLARATION_GENERATOR must define:
#   .generate_declaration(fd, indentation) - write to the given file-like object
#       any generated declaration code. Parameters are the same as for a
#       STATEMENT_GENERATOR.
DECLARATION_GENERATOR = Category()

# An LVALUE_GENERATOR is an object which is allowed to appear at the left-hand
    side
# of an assignment.
# An LVALUE_GENERATOR must define:
#   .generate_lvalue(fd) - must write to the given file-like object any
#       generated code for this lvalue.
LVALUE_GENERATOR = Category()

# An EXPRESSION_GENERATOR is an object which is allowed to appear at the
    right-hand
# side of an assignment.
# An EXPRESSION_GENERATOR must define:
#   .generate_expr(fd) - write to the given file-like object any generated code
#       for this expression.
#   .precedence - must be a non-negative numerical value defining the precedence
#       of this operation compared to other operations. This is used for
#       determnining whether or not to generate brackets around the expression.
#       A higher value of precedence corresponds to a more tightly-binding
#       operation.
#   The precedence values used by tsPyC are:
#       10  assignment
#       20  ternary conditional
#       30  logical or
#       40  logical and
#       50  bitwise or
#       60  bitwise xor
#       70  bitwise and
#       80  == and !=
#       90  >, >=, <, <=
#       100 << and >>
#       110 addition and subtraction
#       120 multiplication, division, modulus
#       130 logical negation, other unary operations
#       140 suffix operations such as function call, array subscript
#       150 never needs brackets (e.g. variable name, already has brackets)
EXPRESSION_GENERATOR = Category()

# A CONDITION is an object which is allowed to appear as a boolean
# condition in statements such as while, if, etc.
#   * It must be a valid EXPRESSION_GENERATOR
```

```
CONDITION = Category(EXPRESSION_GENERATOR)

# An ELSEABLE is an object which is allowed to be followed by an else block.
# An ELSEABLE must have the method:
#    .attach_else(state, elseblock) - must return the object formed from
#        attaching an else block to the object. elseblock is a
#        STATEMENT_GENERATOR.
ELSEABLE = Category()
```

Appendix D

Base Language

This appendix describes the symbols made available as part of tsPyC's base language, and the associated semantics. The base language defines the following symbols:

- `array`—used to define an array type. For instance, an integer array of size three could be defined using `x: array(int, 3)`.
- `break`—used to break out of the currently executing loop. Must be used as the only symbol on a line. The position to which the `break` command will direct flow of control may be defined within custom control structures by defining a label called `__break__`.
- `byte`—integer type representing a number in the range 0 to 255.
- `continue`—used to advance to the next cycle of the currently executing loop. Must be used as the only symbol on a line. The position to which the `continue` command will direct flow of control may be defined within custom control structures by defining a label called `__continue__`.
- `elif`—short for "else if". May only be used immediately following an `if` or `elif` block. Causes a block of code to be executed only if a particular condition is met and none of the immediately previous `if` and `elif` blocks have had their conditions met. Must be followed on the same line by a boolean expression indicating the condition. The `elif` line must be immediately followed by an indented block of statements to be executed when the condition is met.
- `else`—may only be used immediately following an `if` or `elif` block. Causes a block of code to be executed only if none of the immediately previous `if` and `elif` blocks have had their conditions met. Must be the only symbol on its line, and must be immediately followed by an indented block of statements to be executed.
- `false`—boolean constant.
- `float`—floating-point type corresponding directly to the `float` type of the underlying C implementation.
- `function`—keyword used to define functions and function types. An example of use to define function types is: `fn: function(type1, type2 -> resulttype)`. May also be used to define functions, as in the following example:

```
x := function(a: type1, b: type2 -> resulttype)
    function_body
```

- `if`—causes a block of code to be executed only if a particular condition is met. Must be followed on the same line by a boolean expression indicating the condition. This line must be immediately followed by an indented block of statements to be executed when the condition is met.
- `int`—integer base type corresponding directly to the `int` type of the underlying C implementation.
- `pass`—statement that generates no code.

- `program`—the default tsPyC file type. Should be used in the context of `begin program` to separate the preamble from the body of a tsPyC source file.
- `ptr`—used to construct a pointer type, as in `x: ptr(int)`. Also used to take the address of an expression, as in `x = ptr(y)`. A pointer `x` may be dereferenced using the syntax `x.value`.
- `return`—returns a value from a function. Must be followed by an expression representing the value to return.
- `scanf`—convenience wrapper around the C `scanf` function.
- `string`—type representing strings of characters.
- `struct`—used to define `struct` types. Must be followed by an indented block containing lines representing the struct members. For instance:

```
t := struct
     x: int
     y: int
```

- `true`—boolean constant.
- `var`—used to define variables. For instance, x := var(int) defines an int variable. Note that `x: int` is shorthand which is expanded internally to `x := var(int)`.
- `void`—empty type. Used primarily to define functions with a return type but no parameters, as in `function(void -> returntype)`.
- `while`—loop control structure. Must be followed on the same line by a boolean expression which is evaluated at the start of each execution of the loop. This line must be followed by an indented block of statements defining the loop body.

Bibliography

[1] PyInline: Mix other languages directly inline with your Python, November 2004. `http://pyinline.sourceforge.net/`, accessed Mar. 2009.

[2] PyPy[index], November 2008. `http://codespeak.net/pypy/dist/pypy/doc/`, accessed Mar. 2009.

[3] Weave (SciPy.org), May 2008. `http://scipy.org/Weave`, accessed Mar. 2009.

[4] David Beazley. PLY (Python lex-yacc), March 2009. `http://www.dabeaz.com/ply/`, accessed Mar. 2009.

[5] Stefan Behnel and Robert Bradshaw. Cython: C-extensions for Python. `http://www.cython.org/`, accessed Mar. 2009.

[6] Stefan Behnel and Robert Bradshaw. Differences between Cython and Pyrex (Cython v0.11 documentation). `http://docs.cython.org/docs/pyrex_differences.html`, accessed Mar. 2009.

[7] Xing Cai, Hans Petter Langtangen, and Halvard Moe. On the performance of the python programming language for serial and parallel scientific computations. *Scientific Programming*, 13(1):31–56, 2005.

[8] Greg Ewing. About Pyrex. `http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/version/Doc/About.html`, accessed Mar. 2009.

[9] Greg Ewing. Pyrex—a language for writing Python extension modules. `http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/`, accessed Mar. 2009.

[10] Antti Kervinen. Cinpy—C in Python, September 2007. `http://www.cs.tut.fi/~ask/cinpy/`, accessed Mar. 2009.

[11] Armin Rigo. Psyco—introduction. `http://psyco.sourceforge.net/introduction.html`, accessed Mar. 2009.

[12] Armin Rigo. Representation-based just-in-time specialization and the psyco prototype for Python. In *PEPM '04: Proceedings of the 2004 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 15–26, New York, NY, USA, 2004. ACM Press.

[13] Armin Rigo and Samuele Pedroni. PyPy's approach to virtual machine construction. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, pages 944–953, New York, NY, USA, 2006. ACM.

[14] D. S. Seljebotn. Fast numerical computations with Cython. In *Proceedings of the 8th Python in Science Conference*, 2009. Preprint from `http://sage.math.washington.edu/home/dagss/numerical-cython-preprint.pdf`, accessed Oct. 2009.

[15] Thomas A. Standish. Extensibility in programming language design. *SIGPLAN Not.*, 10(7):18–21, 1975.

[16] Gregory V. Wilson. Extensible programming for the 21st century. *ACM Queue*, 2(9):48–57, 2004.

[17] Daniel Zingaro. Modern extensible languages. *Software Quality Research Laboratory*, Oct 2007.