Universitetet i Oslo
Institutt for informatikk

Translating Python to C++ for palmtop software development

Thesis

Tom Simonsen
tomsi@ifi.uio.no

1. November 2005

1 Acknowledgments

Thanks to mom, dad, Lisa and Cecilie for all your love and support. A big thanks to music for keeping me sane, and thanks to Ola Skavhaug for having the patience with me when I was being held up by the music.

Contents

2 Introduction

Writing code in a low-level programming language is considered to be more difficult, more error-prone and more time-consuming than writing the same code in a high-level language. A lot of software is implemented in low-level languages today, because it tends to produce efficient software. If this efficient code could be created through the use of a high-level language, many developers would probably prefer to use these types of languages more frequently.

One way to achieve this kind of efficiency in a high-level language is to produce a tool that translates code from one programming language to another. This tool could turn the code of a high-level language into the code of a low-level language. Java or Python could be translated to C or C++ (May hereafter be referred to as one entity with the use of "C/C++", for convenience.), for instance.

Creating code for more than one platform can also be tedious and time-consuming labor. A lot of time and precious finances could be saved if there existed a tool that could take care of this process automatically. One would only have to develop the code for one platform and run this code through the translation tool, which would generate and output the code for the other platforms. The translated bits of code could then be compiled on each respective system by already existing compilers. With such a tool you could create software with support for numerous systems at the cost of creating the code for only one of them. Once you have created a translation tool that translates from Python to C/C++, it would not be too much work to implement support for more than one platform. The C/C++ source code looks a little different from platform to platform, but the most of the code would remain unchanged in the various system versions.

The creation of such a translation tool is by no means a small task, and several attempts and experiments have been carried out to figure out how it can be done. Attempts have been made to translate the high-level language of Python to the low-level language of C. None of the attempts that I am aware of have been able to solve this complex problem in full, but a few useful techniques and methods by which this problem can be solved can perhaps be deduced from these attempts. In this thesis I will take a look at what has been done in the field of code translation, and I will try to create a scaled-down implementation of a Python to C/C++ translator. I will also make investigations into how this translator could translate graphical user interfaces (GUIs). In this regard I will look into how to translate Python to native C++ code for the Pocket PC 2002 operating system, a tiny version of Microsoft Windows that is designed to run on Palmtop computers. This system does not have a command line utility and is heavily dependent on GUIs to display information on-screen.

2.1 The world of palmtops

The palmtops or hand-held devices are basically computers that are small enough to fit in your hand. Many of these devices have functions and abilities that are similar to what you would expect to find on an average stationary or laptop computer. The palmtops allow you to surf the Internet, read and write e-mail, keep track of your appointments, read and write various documents, listen to music and watch video almost wherever you are. PDAs (Personal Digital Assistants) and, increasingly, mobile phones belong to this category of devices. They all have quite a bit of computational power in spite of their physical appearance and size.

Although the devices are getting more and more advanced and powerful, almost on a monthly basis, they still have a way to go before they actually start to rival the average PC. All of these devices do not have much memory or storage capacity, and the operating systems that run on them are significantly scaled down compared to the full blown Windows or Linux operating systems that run on your typical PC.

There are also a number of different software platforms to be found among the various devices on the market today. The Symbian operating system is available on devices in version 4.0 or 6.0. This is an operating system specially designed for running on mobile phones. Microsoft is trying to compete with Symbian through its Windows CE or Pocket PC system, which is a scaled-down version of the Windows operating system for PCs. Windows CE has been a very popular operating system with the PDA manufacturers, and many PDA models have been designed to run this system. Recently, Microsoft has been able to put Windows CE in a few mobile phone models as well. Palm OS is found on the PDAs made by Palm and is currently in version 5.

The open source community has been able to produce scaled-down and specially adapted versions of the Linux operating system for many PDA models, called Embedded Linux [LinuxDevices.com, 2005]. This can be viewed as an alternative for the consumers that want to install another operating system on their PDAs. Some manufacturers also produce PDAs that are designed for running the Linux platform. The products in the Zaurus series from Sharp are among these.

2.2 The main purpose of the thesis

The main goal of the thesis is to produce a software product that translates Python source code to C++. By especially focusing on the world of palmtops, where there are quite a few significantly different platforms, the aim with such a tool is to achieve platform independence. I will only focus my attention on the translation to one palmtop system, namely Windows CE or, more accurately, the Windows Pocket PC 2002 operating system.

I will not make any attempts at translating the Python code to more than one platform. It is the translation process itself I wish to take a closer look at. It is nor my intention to translate the entire language of Python, since this would be a fairly huge task for one man to execute in the time set aside for completing this thesis. I only wish to translate some basic parts of the language, enough to be able to test the translation engine on a set of relatively simple Python programs. (Such a simple program is translated in the below example.) The most basic parts would be things like creation and assignment of variables, if statements, for and while loops, functions and class structures. In addition to this, I also want to take a look at the possibilities for translating graphical user interfaces (GUIs). I have chosen to work with Pythons `Tkinter` module. I will look at the possibilities for my translation engine to translate Python `Tkinter` code to GUI code for Windows Pocket PC 2002.

Below, I have included an example of what I want to achieve. The short Python program is automatically translated by a translator software tool, to C++. This is a small Python program that takes a text as input and then an integer telling it how many times the text shall be printed.

```
#!/usr/bin/env python

class Print:
    unused_var1 = 0
    unused_var2 = ""
    unused_var3 = 2.333

    def __init__(self):
        print "Object created!"

    def printOut(self, number, string):
        for i in range(number):
            if(i % 5) == 0:
                print "\n" + string
            else:
                print string

    def unusedFunction(self):
        print "This function is never called."

string = raw_input("Type a string: ")
temp_string = raw_input("and the number of times it should be printed: ")
number = int(temp_string)
print_object = Print()
print_object.printOut(number, string)
print "\nString \"" + string + "\" printed " + str(number) + " times."
```

The translation produces two files containing the translated C++ code. The first of them is the code that contains the `main` function.

```
/* File translated Tue Sep 13 15:33:13 2005
 * by Python2C version 0.013. */

#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>

/* The python types translation headers: */
#include "python2clist.h"
#include "python2ctuple.h"
#include "python2cdict.h"

/* The includes generated by the translation: */
#include "Print.h"

char* string;
char* temp_string;
int number;
Print* print_object;
```

```
int main(int argc, char* argv[])
{
  printf("Type a string: ");
  string = (char*) malloc(256);
  scanf("%s", string);
  printf("and the number of times it should be printed: ");
  temp_string = (char*) malloc(256);
  scanf("%s", temp_string);
  number = atoi(temp_string);
  *print_object = Print();
  print_object->printOut(number, string);
   cout << "\nString \"" << string << "\" printed " << number << " times." <<
endl;
  return 0;
}
```

This is the second file that is produced by the translator software. It contains the translated C++ code for the `Print` class.

```
/* File translated Tue Sep 13 15:33:13 2005
 * by Python2C version 0.013. */

#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>

/* The python types translation headers: */
#include "python2clist.h"
#include "python2ctuple.h"
#include "python2cdict.h"

class Print
{
 public:
  int unused_var1;
  char* unused_var2;
  double unused_var3;
  ~Print();
  void printOut(int number, char* string);
  void unusedFunction();
  Print();
};

Print::Print()
{
  unused_var3 = 2.333;
  unused_var2 = "";
  unused_var2 = (char*) malloc(256);
  unused_var1 = 0;
  cout << "Object created!" << endl;
}

void Print::printOut(int number, char* string)
{
  int i;
  for(i = 0; i < number; i += 1)
    {
      if(i % 5 == 0)
      {
        cout << "\n" << string << endl;
      }
      else
      {
        cout << string << endl;
      }
    }
}

void Print::unusedFunction()
{
  cout << "This function is never called." << endl;
}

Print::~Print()
{
}
```

I have chosen to work on a PDA model running the Windows Pocket PC 2002 operating system, called iPAQ, developed by Hewlett-Packard. My task will be to develop a Python to C++ translator for this specific device. Once a basic translation engine has been developed, additions can be made to create a software package that is able to translate Python code to native C++ code for other platforms. This is outside the scope of this thesis, however.

2.2.1 Why bother translating one language to another?

2.2.1.1 The need for speed

One answer to the above question is "the need for speed". This need especially crops up in areas of software development where one needs to produce complex graphics or other heavy computations. Another area is in software development for devices that do not have much computational power or memory storage, such as most of the palmtops.

The palmtops or hand-helds usually have slower processors and storage facilities than the regular desktop or laptop computers. This is mostly due to the share size of these small devices. Sometimes it is not possible to fit the really fast hardware into such a small space, or the battery that is going to power the fast hardware would get too large. It is up to the manufacturers of the device to make the trade-off between the desired computing power and how long the device is going to be able to run without a recharge of its batteries. As a consequence, the hand-held devices that have been developed are usually a result of a compromise between hardware requirements, size requirements and power requirements.

2.2.1.2 Platform independence

Another reason for turning one language into another is the need for platform independence, as mentioned earlier. A problem that software developers often are faced with today, is that the code that works really well while running on one specific platform, may run quite poorly - or possibly not at all - on another. It may be tedious labor to manually convert the original code into something that also works well on another system. If the original code had been written in a platform independent language, all this work could have been saved at the cost of the software running less efficiently. In the field of developing software for hand-held devices, the need for platform independence is definitely felt because of the large number of product models that exist on the market.

To counter this problem, a language called Java has been developed. In this case, the platform independence is achieved through the use of a so-called virtual machine, a program that runs on top of the underlying system and provides an identical interface for all Java programs. The use of the virtual machine turns Java into a language that is not normally preferred when the "need for speed" is felt, as Java can be pretty slow when compared to competitor languages such as C or C++. C and C++ do not use a virtual machines. Regardless of this, Java is currently being used more and more in cellular phones, especially to develop games.

In the area of software development for palmtops, where both the need for speed and platform independence is felt, Java does not really offer a complete solution as yet. As already mentioned, the computational power of these devices is relatively weak, and the memory of these devices do not provide much space for storing programs. Some of these devices may only have one storage area, namely the RAM of the device. This means that these devices store all files and the data of running programs in the same, small memory area, and extreme caution is therefore advised when controlling the way that these programs use this scarce resource. Not paying enough attention to this issue may produce fatal errors from the use of the software.

2.2.1.3 Time is money

Programming in high-level languages is generally perceived as easier and more efficient. The time it takes to get a working prototype of a program up and running is low compared to the time is takes to develop a similar thing in a low-level language, like C or C++. The high-level languages are often used to produce software prototypes. Thus, the final software product is often the result of some kind of a translation process from this prototype, to a software product written in a lower-level language. Sometimes the end result comprises a hybrid of high-level and low-level languages.

It is generally quick and easy to produce a fairly functional program with a GUI in high-level languages. The problem with these languages, when developing for palmtops, is the fact that they run relatively slowly and use too much memory. Developing software for hand-held devices using anything other than C/C++ (or assembly language, for that matter) may turn out software that is too slow to operate well. You have to pay good attention to speed and the way that the software uses memory. C/C++ has utilities for controlling the use of memory in detail (This is not possible in Java or Python, for instance). Software development in C/C++ can be quite time consuming and cumbersome because of the extended time required for testing, as the memory handling is especially error prone.

2.2.2 Approach to the problem

The task at hand is basically to create a Python compiler. What sets my compiler aside from other compilers is the fact that the target language is a programming language, not machine language. To complete this task I will have to dive into the field of compiler theory and compiler construction, and I will have to study the languages I am going to be dealing with. These languages are Python, `Tkinter` (This is a Python module, but the programming style and syntax of this module is slightly different from the Python language.), C++ and resource script (This is a separate scripting language that defines the layout of the Windows CE GUI.). In addition, I will take a look at other attempts at translating Python to C or C++. I might be able to extract some valuable information from such earlier attempts. Information of this kind might help me to avoid some discovered pitfalls and steer me in the right direction.

I will write my Python to C++ translator in the Python language, based on the information that I am able to gather in my research. As I have already mentioned; this will not be a full-blown Python to C++ translator software that is able to translate every element of the Python language into fully functional, error-free C++ code. The development will go on until I feel like I have covered enough ground to be able to draw my conclusions and answer my questions. When this point has been reached, I will evaluate the entire project and the resulting software to find what has been learned.

2.3 A brief comment on the issue of object-orienting

Python and C++ are object oriented languages in the sense that they support all language concepts that are typical for languages of this type, according to the most widespread opinions. These concepts are classes, data encapsulation, inheritance and polymorphism. There is an ongoing debate as to whether C++ is an object-oriented language in the true sense of the term, mostly because of its backward compatibility with C. This allows programs to be written in C++ that are not using any of the above mentioned concepts or features. From this perspective, Python can also be viewed as “not entirely object-oriented” since it allows programs to be written in a functional manner, without the explicit use of objects or classes.

I use the term object-oriented loosely anywhere it occurs in this entire thesis. Any language that supports the above mentioned features is in my view an object-oriented language.

2.4 Python

Python is often referred to as a scripting language and is often mentioned in the same sentence as and compared to the language Perl, which is used widely by software developers. Both Python and Perl belong to a group of “very high-level languages” [Langtangen, August 2002] where you can typically achieve a lot by writing relatively small amounts of code, especially when compared to lower-level languages like Fortran, C/C++ or Java. The scripting languages are often used for crunching large amounts of data, building prototype software, creating graphical user interfaces or connecting programs written in other languages together, to form new and powerful software bundles.

The programming style and versatile syntax of these languages often offer new possibilities compared to other languages. The software is created faster and requires less lines of code. Python has an advantage over Perl when it comes to the user-friendly and clean syntax. When prototype software is created in Python, the syntax of the source code can be relatively easily understood by people that are not familiar with the language. The language is also easy to learn by programmers.

2.4.1 The history of Python

In 1989 a programmer named Guido Van Rossum decided he would try to design a new programming language. Van Rossum was working for Stichting Mathematisch Centrum in the Netherlands on the Amoeba distributed operating system group. [Python.org, May 20, 2004] While working on this project Van Rossum was looking for a scripting language to ease his work tasks. He wanted a language that had a syntax similar to a language called ABC, but with access to the Amoeba system calls. Having some experience with using the Modula-2+ language, a language in the Pascal family [Digital Equipment Corporation, September 1999], Van Rossum decided to have a talk with the designers of Modula-3. Some of Pythons features are inspired by this particular language, especially the syntax and semantics used for exception handling.

Python version 0.9.0 was released in 1991, getting its special name from a not entirely unknown British comedy series from the seventies, called "Monty Python's Flying Circus". The Python Software Foundation (PSF) was formed in 2001, a non-profit organization created to own the Python-related intellectual property. This organization continues the open source development and release of the Python development software, currently in version 2.4.2 (released on September 28, 2005).

2.4.2 The Python programming language

Python is an object oriented scripting language with a clean and user-friendly syntax that is easy to use and understand [Langtangen, August 2002]. It is available on a large number of platforms, so that the Python code that you write on a Linux platform will run on Windows without any alterations, in most cases. This is also the case with the Python code that makes up the GUI, if you use one of the GUI modules that support various platforms. Python has bindings to the Tk library for GUI programming, making it very fast and easy to create GUIs. The fact that Python is object oriented makes for faster and more flexible GUI programming.

Most languages like C, C++ or Java are statically and strongly typed. Every variable you create has to be of a certain type. Strong typing is present in these languages to minimize the number of errors in a program and to pair the right data with the right variables and functions. Python is dynamically typed and you do not have to declare the types explicitly. A variable is simply created without a type in the source code. This does not mean that types do not exist in Python, however. Python is actually more strongly typed than C/C++. When a variable is created and assigned a value of some sort, the variable will get the same type as the type of the value assigned to it. The type of this variable can also change during execution. A Python variable can be assigned a value of a different type at any time, which will change the type of the variable itself. If this were to occur in a C/C++ or Java program, the compiler would typically report the “wrong type in assignment” error. In a similar fashion, the return type of a Python function is not declared, making it possible for a Python function to return values of more than one type. The types of the function arguments are not declared either, making it possible to send any type of data into a function.

The fact that Python is not statically typed makes it a more flexible language with a larger potential for code reuse. Where you would have to use multiple functions in C/C++ or Java to be able to send different types of data to a function, you may only have to use one in Python. The run-time environment in Python continually checks for bugs related to the types of variables and return types of functions, at the cost of the programs' efficiency and the time it takes to execute the code.

There is no need to worry about memory management in Python since this is taken care of by the run-time environment. In C and C++ you have to worry about allocating and freeing space in memory for variables and data structures. A fair amount of time of the development process has to be spent on paying attention to this issue, and many errors usually arise from bugs in the memory management code. The elimination of the need for a programmer to worry about this is a great time-saving feature. The run-time environment of Python takes care of the allocation of memory and the "garbage collection". This is done with a loss of efficiency, of course, because the garbage collection algorithm has to check for unreferenced variables and data structures and delete these, at given intervals to free memory.

Python code is interpreted, not compiled. This means that each statement in the code is turned into machine instructions as the program is running. With compiled languages you have to create an executable file from the source code using a compiler program. Then this file typically has to be executed by the operating system to run the program. When a program is interpreted, the program is not run directly by the operating system. Instead, the code is interpreted and run by an interpreter program. This program can check for errors during run-time and add features like the automatic garbage collection.

Python is a very dynamic programming language. It is possible to generate new code during run-time and execute it. It is also possible to make changes to a class while the program is running, by adding variables that were not present in the original code.

2.4.3 Python GUI modules

Python makes GUI programming fairly easy when compared to other programming languages that may support these kinds of features, like C++ or Java. In Python you have a series of GUI toolkits or modules to chose from that can be used depending on what sort of graphical interface you would like to create for your program. None of these toolkits are actually a part of the Python language itself, but are instead additional utilities that can be installed at will. All Python GUI toolkits have been written as extensions to the Python language. The Python GUI toolkits might have a syntax that differs from the sort of syntax you normally associate with Python, depending on how the toolkits are meant to be used and how they have been developed and designed. The programming style may also differ from the general programming style of the Python language. New modules can also be written trough what has been termed *Python Extension Programming* [Rossum and Drake, June 22, 2001].

- **`Tkinter`**

  The `Tkinter` module is an interface to the Tk GUI toolkit that used to be developed by Sun Labs and is currently being developed by Scriptics. It is one of the oldest Python GUI toolkits, supporting the Windows, Unix, Linux and - to a certain degree - also the Mac OS. It is easy to use, but lacks the ability to create GUIs with a real look and feel of any native platform GUI (if you ignore the 8.0 release and later releases). The Tk interface module also contains a number of other Python modules of which the `Tkconstants` is the most important. Both `Tkinter` and `Tkconstants` import one another so it does not matter which one you actually put in an import statement in the Python source code. [Lundh, 1999]

- **`Tix`**

`Tix` is short for “Tk Interface eXtension”, and, as the name suggests, it is meant to be used with the existing `Tkinter` module to extend it and enhance GUIs with an additional rich set of graphical elements, called *widgets* in Python terminology. Both `Tkinter` and `Tix` are included in the releases of Python from Python.org. One advantage of the `Tix` module over `Tkinter` is that many of the of the features in `Tix` have been implemented in native source code for each platform. This enhances the performance of the GUIs and improves the look and feel of them from a user-perspective. [Python.org, November 29, 2004]

- **`PyQt`**

  This Python GUI module is based on the C++ GUI toolkit made by Troll Tech, called Qt. It supports the Windows, Unix, Linux and Mac OS X platforms and binds the complete Qt library, with a few exceptions. Apposed to the `Tkinter` module this GUI toolkit makes it possible to create the look and feel of the native platform GUI on a number of systems, and it comes with a fairly large inventory of advanced GUI features. [Rempt, January 1, 2002]

- **`wxPython`**

  This module is based on the wxWidgets GUI toolkit, implemented in C++. `wxPython` interfaces with the native C++ GUI toolkit of each platform, making it possible to program native platform GUIs with Python. wxWidgets, originally developed by the Artificial Intelligence Applications Institute at the University of Edinburgh, provide the programmer with GUI facilities and more for several different platforms. Version 2 of the software supports the Windows, Unix and Linux platforms. Mac OS support is also underway. [Smart, Roebling, Zeitlin, Dunn, et al, October 2004]

- **`FXPy`**

  `FXPy` is based on another cross-platform GUI toolkit implemented in C++, called FOX, this GUI toolkit is one of the least known and least used by programmers out there. The main focus of this toolkit is execution speed, and in addition to being a GUI toolkit it also acts as an interface to OpenGL and Mesa facilities, making it easy to do 3D-modeling. This module currently supports the Windows, Unix and Linux platforms. [Johnson, 2002]

- **`PyGTK`**

  The `PyGTK` toolkit mainly supports the Unix and Linux platforms and is a wrapper for the GTK+ or Gimp toolkit. The GTK+ toolkit was, among other things, used to create the popular GNOME desktop interface for Linux. `PyGTK` was never intended for any cross-platform development, but it has recently been given some support for Windows and Mac OS X. It provides a comprehensive set of widgets and has Unicode and bidirectional text support. [The GNOME Project, 2003]

- **`Pythonwin`**

  This module is a binding to the Microsoft Foundation Class (MFC) library, and thus this GUI toolkit only has support for the Windows platform. `Pythonwin` consists of the poorly documented module `win32ui`, that provide access to the raw MFC classes. For every MFC object there usually exists a `win32ui` counterpart. Functions in the `win32ui` classes can be overridden using subclassing, which provides the programmer with a flexible, powerful and object oriented tool that is easier to use than the C++ toolkit it is based upon. [Hammond and Robinson, January 2000]

2.5 C++

C++ can be regarded as an extended version of C, with classes and other features that qualify a language for being object-oriented. It is entirely backward compatible with C and was originally designed with this in mind. The result of this compatibility feature is a kind of hybrid language that is both object-oriented and functional (meaning “not object oriented”, like C or Fortran). It is possible to write programs in C++ that are entirely functional and do not make use of any of the language's object-oriented features.

Since C++ was created in the eighties, it has become very widespread and popular in its use. It has become the major programming language used for developing applications for almost any platform or operating system today. The language is very efficient, just as its predecessor, and there is almost no system that does not have a C++ compiler for itself. Its efficiency comes to good use when developing software with GUIs or 3D-graphics.

2.5.1 The history of C++

The basis for C++, C, was a functional programming language developed by Bell Labs from around 1969 to circa 1973. With Bell Labs being the developer of the UNIX operating system, C was originally developed for and implemented on this platform. Ninety percent of the original UNIX operating system was written in C.

In 1989 the C programming language was standardized by ANSI (American National Standard Institute). This was almost out of pure necessity, since there existed quite a number of confusingly different versions of the language at the time, especially before the ANSI committee started the standardization work in the early eighties.

Prior to 1983 Bjarne Stroustrup at Bell Labs wrote an extension to C called “C with classes” that was very much inspired by the object oriented Simula programming language. The term C++ was first used in 1983. This new programming language was further developed for the UNIX system by AT&T under the leadership of Bjarne Stroustrup, with the aim to create an object-oriented extension of C that would allow existing C code to be used whenever possible. The first commercial release of the language and its manual was published in 1985.

In 1998 C++ became a standardized language when an ANSI/ISO (American National Standard Institute/International Organization for Standardization) committee adopted a worldwide uniform language specification. This standard was amended in 2002, creating the second version that is still in effect, to be followed by developers of compilers for this language worldwide. [The C++ Resources Network, 2000]

2.5.2 The C++ programming language

C++ is a statically typed, object-oriented, backward compatible hybrid-language. A C++ program can be compiled on almost any platform without any alterations. At least this is true in most cases where the program has been written using the basic C++ libraries (equivalent to “modules” in Python terminology). If some special libraries have been used in a program this might not be the case, however. For Windows GUI programming for instance, there exists a number of different libraries that can be included into a C++ program. Programs that use these facilities will typically not compile without errors on a UNIX system, for instance. When using only the basic C++ libraries, compiler errors may arise due to the way that different processors handle and represent the data. The bit resolution of various types of data may vary across different systems and cause problems.

New toolkits for this programming language is being developed all the time. There exists especially a lot of different toolkits that have been designed for creating GUIs or 3D-graphics. C++ is very popular for creating 3D-graphics because of its efficiency and memory management features that allow you to write your own memory handling routines and make them as efficient as possible. C++ is a language that is compiled and has no run-time environment facilities. There is no extra processing power being used on checking the validity of a program during execution and there is no built-in garbage-collection, unless you build these sort of things into the software yourself.

2.6 Windows programming

The Windows programming environment differs from the environments found on other development platforms, such as Linux. All is more centered around the users actions and the graphical user interface of a program. On Linux, the programmer's focus is more on firstly creating the program and secondly creating a GUI that wraps around it. The Windows environment turns this way of viewing the development process on its head and forces the programmer to firstly think of the way the program is presented to the user. The GUI may be created first, and the underlying program is written to serve the GUI. As with other platforms, you can of course create the underlying program first. The software can always be tested by using the Windows command line, and the GUI can be created later. Never the less, the Windows platform is focused on GUIs rather than the old command line and its textual interface. The command line is very seldomly used by Windows applications and the command line function is more hidden away from the users of the system. On Linux, the command line still gets a lot of attention and is still at the heart of the platform. Many Linux programs do not have GUIs, although the number of GUI programs for this system is increasing rapidly.

2.6.1 The basics of the Windows platform

2.6.1.1 The windows

A Windows program consists of various visual components called *windows*. A *window* can be any kind of visual GUI object from a simple click-to-push button to an entire window that can contain multiple child windows. Each window has an *appearance* associated with it, defining how the window looks on-screen to the user. The data that are associated with a window are called *properties* in Windows terminology. The properties may be the text on the face of a button or the options in a window menu. All windows are able to respond to *events*, or actions that are typically generated by the user clicking on some visual element in the user interface. Windows programs are referred to as being *event-driven*. Each individual window listens for events and determines what action to take, based on what kind of event that occurs.

Each type of window is called a window class, and each class has a predefined appearance, property set and action set. A normal Windows program will typically make use of many different kinds of these classes. It is possible to make your own window classes from scratch or by enhancing an already existing class, by using inheritance for instance.

When a window in a program creates or contains other windows, this main window is referred to as the *parent window* and the windows created by the main window are referred to as *child windows*. A child window can also be the parent of other windows that will be its child windows. Both a parent and a child window can respond to events without relying on each other, but if a parent window is destroyed, closed, hidden, moved or disabled, then so is the child window and what ever child windows it might have. A child window can also send messages to its parent when an event occurs, and let it be up to the parent window to take the appropriate action.

2.6.1.2 Handles, resources and messages

In a Windows program there are two fundamental elements: *handles* and *resources*. A *resource* is a component (usually a graphical component) that has some kind of function in the program. The resource identifier in the program is called a *handle*. To be able to use a resource in a Windows program you have to acquire the handle for it. Each window in an application has a unique window handle which is identified with the `HWND` data type. A handle can be obtained by calling one of the `CreateWindow`, `GetWindow` or `FindWindow` functions.

Interaction in a Windows program is achieved through sending messages. These messages can be used to interact with the Windows system or with other components of the same program. Such a message is represented by an integer and is passed on through using the `SendMessage` function.

The message processing involves the message queue, the message loop and the window procedure. All events that occur are detected by the Windows system, which passes the messages along to the right application and puts it in the application's message queue. Every Windows program has a message queue created for it by the Windows system when the program is started. The application receives the messages sent to it by polling or removing the messages from this queue, using the `GetMessage` or `PeekMessage` functions. Messages that are received must be forwarded to the proper window inside the application. In a similar fashion, messages can be placed by using the `PostMessage` function.

The message loop is a series of statements that is responsible for polling the message queue, and sending the messages to the right window. A window procedure is a function that reacts and ignores specific Windows messages, depending on how it is programmed. It is a *callback function*, which means that it does not have to be called directly. The Windows operating system will call this function when necessary. In addition to handling the messages, this procedure controls the appearance and behavior of all windows of the class that it belongs to. In many cases, this procedure does not have to be written as there already exists predefined window procedures for the common window classes.

2.6.1.3 The resource script

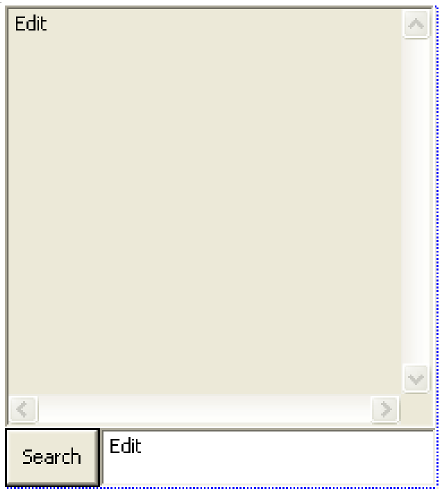
The *resource script* is a scripting language for defining the various GUI elements in a Windows program. The visual elements of each graphical component in the Windows program is tied to the rest of the program through this scripting language. Each graphical component in Windows must have its own ID that uniquely distinguishes it from all other graphical components in the program. The IDs are simply unique names that are used to refer to the graphical components both in the C++ code and in the resource script. The resource script is usually kept in its own file with a ".rc" extension. This file is tied together with the rest of the code during compilation by a development software package, such as Microsoft's Visual C++ or eMbedded Visual C++ (for palmtop software development only).

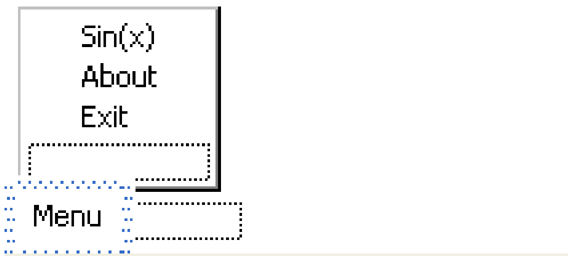
A feature that is commonly found in these types of software packages is that the software packages create and maintain the resource script for you. A programmer does not have to worry about it or to write any of it. The software package will generate all the resource script code as the graphical components are created by the programmer. See chapter 2.6.1.4 for an example of what the resource script code looks like and how it is used as a part of the source code to build the GUI.

2.6.1.4 A Windows CE program example

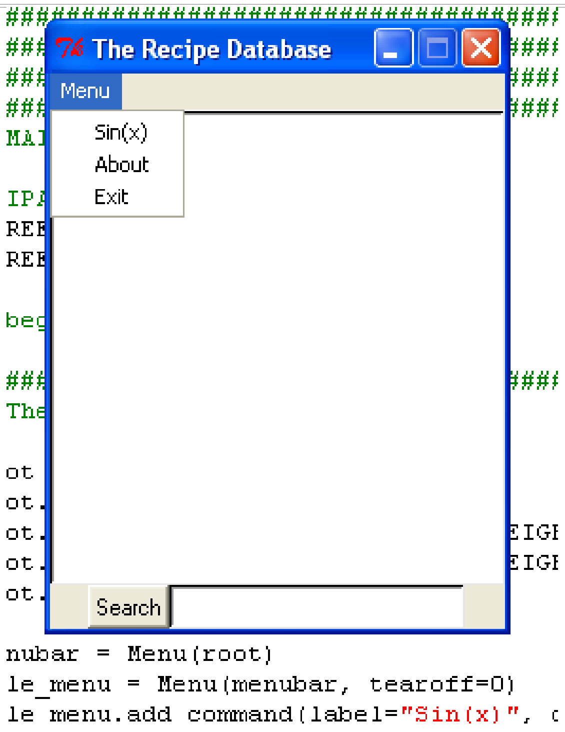
Below are screenshots and the complete source code for a simple Windows CE program. The source code illustrates the way the C++ code and the resource script are combined to create the resulting program. An equivalent Python example has also been included to exemplify the differences between these programming languages. The differences between the Python source code and the Windows CE source code illustrates the challenges faced by a programmer who is trying to create a translator program that translates code from the former to the latter.

These are two screenshots taken when the source code has been loaded into the eMbedded Visual C++ software. This serves to illustrate what the executable program will look like on a Pocket PC device. The first screen shot shows the text fields and the "search" button, the second shows the program's file menu. The menu bar of an application is usually located at the bottom of the screen on the Pocket PC platform, as opposed to being placed at the top of a window on the Windows 95/98/ME/NT/2000/XP or Linux platforms.





Below is an example of how the same program might look when created using Python source code. The screenshot was taken when running the Python code using the Python IDLE software from Python.org.



```
nubar = Menu(root)
le_menu = Menu(menubar, tearoff=0)
le_menu.add_command(label="Sin(x)", c
```

This is the resource script code, contained in the file `TestGUI.rc`*, that defines the layout of the GUI of the simple Windows program. This code was written by hand and has not been generated by any kind of development software.*

```
#include <windows.h>
#include "resource.h"

#if !defined(UNDER_CE)
#define UNDER_CE _WIN32_WCE
#endif

#if defined(_WIN32_WCE)
```

```
      #if !defined(WCEOLE_ENABLE_DIALOGEX)
            #define DIALOGEX DIALOG DISCARDABLE
      #endif
      #include <commctrl.h>
      #define  SHMENUBAR RCDATA
      #if defined(WIN32_PLATFORM_PSPC) && (_WIN32_WCE >= 300)
            #include <aygshell.h>
            #define AFXCE_IDR_SCRATCH_SHMENU  28700
      #else
            #define I_IMAGENONE     (-2)
            #define NOMENU 0xFFFF
            #define IDS_SHNEW 1

            #define IDM_SHAREDNEW 10
            #define IDM_SHAREDNEWDEFAULT 11
      #endif // _WIN32_WCE_PSPC
      #define AFXCE_IDD_SAVEMODIFIEDDLG 28701
#endif // _WIN32_WCE

#ifdef IDC_STATIC
#undef IDC_STATIC
#endif
#define IDC_STATIC (-1)

/////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_CALC DIALOG DISCARDABLE  0, 0, 156, 81
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU // See STYLE Statement
in Visual C++ Help for all options!!!
CAPTION "Sin(x)"
FONT 8, "System"
BEGIN
    DEFPUSHBUTTON   "OK", IDOK, 19, 60, 48, 14
    PUSHBUTTON      "Cancel", IDCANCEL, 90, 60, 49, 14
    LTEXT           "Compute sin(x)", IDC_STATIC, 53, 7, 50, 8
    LTEXT           "sin(", IDC_STATIC, 14, 31, 12, 10
    EDITTEXT        IDC_CALCEDIT1, 26, 29, 40, 15, ES_AUTOVSCROLL | WS_VSCROLL
    LTEXT           ") =", IDC_STATIC, 68, 31, 10, 10
    EDITTEXT        IDC_CALCEDIT2, 83, 29, 59, 15, ES_AUTOHSCROLL | ES_READONLY
END

/////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_SEARCH DIALOG DISCARDABLE  0, 0, 161, 166
STYLE WS_POPUP | WS_VISIBLE
FONT 8, "System"
BEGIN
    EDITTEXT        IDC_SEARCHEDIT1, 0, 0, 160, 145, ES_MULTILINE |
ES_AUTOHSCROLL |
                    ES_READONLY | ES_WANTRETURN | WS_VSCROLL | WS_HSCROLL
    DEFPUSHBUTTON   "Search", IDC_SEARCHBUTTON1, 0, 145, 35, 20
    EDITTEXT        IDC_SEARCHEDIT2, 35, 145, 125, 20, ES_AUTOHSCROLL
END


/////////////////////////////////////////////////////////////////////////////
//
```

```
// Data
//

IDM_MENU SHMENUBAR MOVEABLE PURE
BEGIN
    IDM_MENU, 1,
    I_IMAGENONE, IDM_MAIN, TBSTATE_ENABLED,
    TBSTYLE_DROPDOWN | TBSTYLE_AUTOSIZE, IDS_MENU, 0, 0,
END

/////////////////////////////////////////////////////////////////////////////
//
// Menubar
//

IDM_MENU MENU DISCARDABLE
BEGIN
    POPUP "Menu"
    BEGIN
        MENUITEM "Sin(x)", IDM_CALC
        MENUITEM "About", IDM_ABOUT
        MENUITEM "Exit", IDM_EXIT
    END
END

/////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_ABOUT DIALOG DISCARDABLE  0, 0, 140, 63
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
EXSTYLE 0x80000000L
CAPTION "About TestGUI v1.0"
FONT 8, "System"
BEGIN
    ICON IDI_APPLICATION, IDC_STATIC, 11, 17, 21, 20
    LTEXT "TestGUI v1.0", IDC_STATIC, 40, 10, 70, 8, SS_NOPREFIX
    LTEXT "(C) Tom Simonsen, 2004", IDC_STATIC, 40, 25, 84, 8
    PUSHBUTTON "OK", ABOUT_OKBUTTON, 44, 42, 50, 14
END

/////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_APPLICATION ICON DISCARDABLE "TestGUI.ICO"

/////////////////////////////////////////////////////////////////////////////
//
// String Table
//

STRINGTABLE DISCARDABLE
BEGIN
    IDS_MENU "Menu"
END
```

Below is the contents of the `resource.h` file that is included by the ".rc" file and the main C++ code. This file defines the values of the unique IDs given to each graphical component. This file is also normally generated by development software, together with the resource script code. In this case the code was written by hand.

```
#define IDM_MENU                                101
#define IDM_EXIT                                102
#define IDM_ABOUT                               103
#define IDD_ABOUT                               104
#define ABOUT_OKBUTTON                          105
#define IDI_APPLICATION                         106
#define IDM_MAIN                                107
#define IDS_MENU                                108
#define IDD_CALC                                109
#define IDC_CALCEDIT1                           110
#define IDC_CALCEDIT2                           111
#define IDD_SEARCH                              112
#define IDC_SEARCHEDIT1                         113
#define IDC_SEARCHEDIT2                         114
#define IDC_SEARCHBUTTON1                       115
#define IDM_CALC                                116
```

Below is the main C++ source code for the program, contained in a file called `TestGUI.cpp`. Here you can see how the IDs are used to check the identity of the graphical component where an event has occurred.

```
// "Standard" includes:
#include <windows.h>
#include <commctrl.h>
#include <aygshell.h>
#include "resource.h"
#include <math.h>
#include <stdlib.h>
#include <oleauto.h>

// Forward declarations:
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
HWND CreateMenuBar(HWND hWnd);
LRESULT CALLBACK About(HWND hDlg, UINT uMessage, WPARAM wParam, LPARAM lParam);
LRESULT CALLBACK Calc(HWND hDlg, UINT uMessage, WPARAM wParam, LPARAM lParam);
LRESULT CALLBACK SearchDialog(HWND hDlg, UINT uMessage, WPARAM wParam, LPARAM
lParam);
void calculate(HWND hDlg);

// Globals:
TCHAR strClassName[] = _T("TestGUI");
TCHAR strTitle[] = _T("TestGUI");
HINSTANCE hInst;
HWND hwndCB;
char buffer[256];

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInst, LPTSTR lpCmdLine,
int nCmdShow)
{
      HWND hWnd;
      MSG Msg;

      WNDCLASS WndClass;
      WndClass.hInstance = hInstance;
      WndClass.lpszClassName = strClassName;
      WndClass.lpfnWndProc = WndProc;
      WndClass.style = 0;
      WndClass.hIcon = LoadIcon(NULL, MAKEINTRESOURCE(IDI_APPLICATION));
      WndClass.hCursor = LoadCursor(NULL, IDC_ARROW);
      WndClass.lpszMenuName = NULL;
      WndClass.hbrBackground = (HBRUSH) GetStockObject(WHITE_BRUSH);
      WndClass.cbClsExtra = 0;
      WndClass.cbWndExtra = 0;

      hInst = hInstance;
      if(!RegisterClass(&WndClass))
      {
            MessageBox(NULL, _T("Error registering class."), NULL,
MB_ICONEXCLAMATION|MB_OK);
            return(0);
      }
      hWnd = CreateWindow(strClassName, strTitle, WS_VISIBLE, CW_USEDEFAULT,
                              CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
NULL,
                              NULL, hInstance, NULL);
      if(!hWnd)
      {
            MessageBox(NULL, _T("Error creating window."), NULL,
MB_ICONEXCLAMATION|MB_OK);
            return(0);
      }
      ShowWindow(hWnd, nCmdShow);
      UpdateWindow(hWnd);

      while(GetMessage(&Msg, NULL, 0, 0))
      {
            TranslateMessage(&Msg);
            DispatchMessage(&Msg);
      }

      return(Msg.wParam);
}

LRESULT CALLBACK WndProc(HWND hWnd, UINT uMessage, WPARAM wParam, LPARAM lParam)
{
      //HDC DC;
      int wmId, wmEvent;

      switch(uMessage)
      {
      case WM_COMMAND:
            wmId = LOWORD(wParam);
            wmEvent = HIWORD(wParam);
            switch(wmId)
            {
            // The menu item "Sin(x)":
            case IDM_CALC:
                  DialogBox(hInst, (LPCTSTR) IDD_CALC, hWnd, (DLGPROC) Calc);
                  break;
            // The menu item "About":
```

```
            case IDM_ABOUT:
                  DialogBox(hInst, (LPCTSTR) IDD_ABOUT, hWnd, (DLGPROC) About);
                  break;
            // The menu item "Exit":
            case IDM_EXIT:
                  if(MessageBox(hWnd, _T("Are you sure you want to quit?"),
_T("Confirmation"),
                   MB_ICONQUESTION|MB_YESNO|MB_DEFBUTTON1) == IDYES)
                  DestroyWindow(hWnd);
            default:
                  return DefWindowProc(hWnd, uMessage, wParam, lParam);
            }
            break;
      case WM_CREATE:
            hwndCB = CreateMenuBar(hWnd);
            CreateDialog(hInst, (LPCTSTR) IDD_SEARCH, hWnd, (DLGPROC)
SearchDialog);
            break;
      case WM_CLOSE:
            if(MessageBox(hWnd, _T("Are you sure you want to quit?"),
_T("Confirmation"),
               MB_ICONQUESTION|MB_YESNO|MB_DEFBUTTON1) == IDYES)
               DestroyWindow(hWnd);
      case WM_DESTROY:
            CommandBar_Destroy(hwndCB);
            PostQuitMessage(0);
            break;
      default:
            return(DefWindowProc(hWnd, uMessage, wParam, lParam));
      }
      return(0);
}

HWND CreateMenuBar(HWND hWnd)
{
      SHMENUBARINFO mbi;

      memset(&mbi, 0, sizeof(SHMENUBARINFO));
      mbi.cbSize = sizeof(SHMENUBARINFO);
      mbi.hwndParent = hWnd;
      mbi.nToolBarId = IDM_MENU;
      mbi.hInstRes = hInst;
      mbi.nBmpId = 0;
      mbi.cBmpImages = 0;

      if (!SHCreateMenuBar(&mbi))
            return NULL;

      return mbi.hwndMB;
}

// Message handler for the About dialog:
LRESULT CALLBACK About(HWND hDlg, UINT uMessage, WPARAM wParam, LPARAM lParam)
{
      SHINITDLGINFO shidi;

      switch(uMessage)
      {
      case WM_INITDIALOG:
            shidi.dwMask = SHIDIM_FLAGS;
            shidi.dwFlags = SHIDIF_DONEBUTTON | SHIDIF_SIPDOWN |
SHIDIF_SIZEDLGFULLSCREEN;
            shidi.hDlg = hDlg;
            SHInitDialog(&shidi);
            return TRUE;
      case WM_COMMAND:
            if (LOWORD(wParam) == ABOUT_OKBUTTON)
            {
                  EndDialog(hDlg, LOWORD(wParam));
                  return TRUE;
            }
            break;
      }
    return FALSE;
}

// Message handler for the Search dialog.
LRESULT CALLBACK SearchDialog(HWND hDlg, UINT uMessage, WPARAM wParam, LPARAM
lParam)
{
      SHINITDLGINFO shidi;

      switch (uMessage)
      {
            case WM_INITDIALOG:
                  shidi.dwMask = SHIDIM_FLAGS;
                  shidi.dwFlags = SHIDIF_DONEBUTTON | SHIDIF_SIPDOWN |
SHIDIF_SIZEDLGFULLSCREEN;
                  shidi.hDlg = hDlg;
                  SHInitDialog(&shidi);
                  return TRUE;
            case WM_COMMAND:
                  if(LOWORD(wParam) == IDC_SEARCHBUTTON1)
                  {
                        SetDlgItemText(hDlg, IDC_SEARCHEDIT1, _T("Search..."));
                        return TRUE;
                  }
                  return TRUE;
      }
    return FALSE;
}

// Message handler for Window Contents.
LRESULT CALLBACK Calc(HWND hDlg, UINT uMessage, WPARAM wParam, LPARAM lParam)
{
      SHINITDLGINFO shidi;

      switch (uMessage)
      {
            case WM_INITDIALOG:
                  // Create a Done button and size it.
                  shidi.dwMask = SHIDIM_FLAGS;
                  shidi.dwFlags = SHIDIF_DONEBUTTON | SHIDIF_SIPDOWN |
SHIDIF_SIZEDLGFULLSCREEN;
                  shidi.hDlg = hDlg;
                  SHInitDialog(&shidi);
                  SendDlgItemMessage(hDlg, IDC_CALCEDIT1, EM_LIMITTEXT,
sizeof(buffer), 0);
                  return TRUE;
            case WM_COMMAND:
                  if (LOWORD(wParam) == IDOK)
                  {
                        calculate(hDlg);
                        return TRUE;
                  }
                  else if (LOWORD(wParam) == IDCANCEL)
```

```
                    {
                          EndDialog(hDlg, LOWORD(wParam));
                          return TRUE;
                    }
                    break;
      }
    return FALSE;
}

void calculate(HWND hDlg)
{
      double number = 0;
      GetDlgItemText(hDlg, IDC_CALCEDIT1, (unsigned short *)buffer,
sizeof(buffer));
      number = atof(buffer);
      number = sin(number);
      wsprintf((unsigned short *)buffer, _T("%f"), number);
      SetDlgItemText(hDlg, IDC_CALCEDIT2, (unsigned short *)buffer);
}
```

Below is the Python code for the equivalent GUI program. This code was written in a "windowsy" manner. Writing the code in this way will probably make it easier for a translation program to translate it to Windows CE code. The creation of all the GUI objects (widgets) is done statically, in the main code. No graphical components are created dynamically. A window that is not to be displayed at this point in the execution is created and then hidden from view using the `withdraw` *function. It can be displayed, when the time comes, by using the* `deiconify` *function.*

```
#!/usr/bin/env python

from Tkinter import *
from math import *

################################################################################
################################################################################
################################################################################
################################################################################
# Code that is not concerned with layout:

def calculate():
    global calc_entry1, calc_entry2
    try:
        number = float(calc_entry1.get())
        calc_entry2.delete(0, "end")
        calc_entry2.insert(0, str(sin(number)))
    except:
        popup1.deiconify()
        calc_entry1.delete(0, "end")

################################################################################
# Message handlers:

# Handler for the "Search" button:
def handler_search_button():
    global text_area
    text_area.delete("1.0", "end")
    text_area.insert("insert", "Search...")
```

```
# Handler for the "Sin(x)" menu item:
def handler_calc():
    global calc
    calc.deiconify()

# Handler for the About menu item:
def handler_about():
    global about
    about.deiconify()




################################################################################
################################################################################
################################################################################
################################################################################
# MAIN CODE:

# IPAQ resolution:
SCREEN_WIDTH = 240
SCREEN_HEIGHT = 320

# begin gui

################################################################################
# The main window:

root = Tk()
root.title("The Recipe Database")
root.maxsize(SCREEN_WIDTH, SCREEN_HEIGHT)
root.minsize(SCREEN_WIDTH, SCREEN_HEIGHT)
root.resizable(0,0)

menubar = Menu(root)
file_menu = Menu(menubar, tearoff=0)
file_menu.add_command(label="Sin(x)", command=handler_calc)
file_menu.add_command(label="About", command=handler_about)
file_menu.add_command(label="Exit", command=root.quit)
menubar.add_cascade(label="Menu", menu=file_menu)
root.config(menu=menubar) # Display menu.

top_frame = Frame(root, width=SCREEN_WIDTH, height=250)

bottom_frame = Frame(root, width=SCREEN_WIDTH, height=40)

vertical_scrollbar = Scrollbar(top_frame)
horisontal_scrollbar = Scrollbar(top_frame)
text_area = Text(top_frame, yscrollcommand=vertical_scrollbar.set,
xscrollcommand=horisontal_scrollbar.set)
text_area.config(bg="white")
text_area.place(relwidth=1, relheight=1)

text_field = Text(bottom_frame, height=1)
text_field.config(bg='white', width=25, setgrid=0)
text_field.pack(side='right', fill='both');

search_button = Button(bottom_frame, text="Search",
command=handler_search_button)
search_button.pack(side='right');

top_frame.pack(side='top')
```

```
bottom_frame.pack(side='bottom')

################################################################################
# The "Sin(x)" dialog:

calc = Toplevel()
calc.withdraw()
calc.title("Sin(x)")
calc.maxsize(SCREEN_WIDTH, SCREEN_HEIGHT)
calc.minsize(SCREEN_WIDTH, SCREEN_HEIGHT)
calc.resizable(0,0)

calc_text1 = Label(calc, text="Compute sin(x)")
calc_text1.pack(side="top")

calc_frame1 = Frame(calc)
calc_frame2 = Frame(calc)

calc_text2 = Label(calc_frame1, text="Sin(")
calc_text2.pack(side="left")

calc_entry1 = Entry(calc_frame1)
calc_entry1.pack(side="left")

calc_text3 = Label(calc_frame1, text=") = ")
calc_text3.pack(side="left")

calc_entry2 = Entry(calc_frame1)
calc_entry2.pack(side="left")

calc_button1 = Button(calc_frame2, text="OK", command=calculate)
calc_button1.pack(side="left")

calc_button2 = Button(calc_frame2, text="Cancel", command=calc.withdraw)
calc_button2.pack(side="left")

calc_frame1.pack(side="top")
calc_frame2.pack(side="top")

################################################################################
# The About dialog:

about = Toplevel()
about.withdraw()
about.title("About")
about.maxsize(SCREEN_WIDTH, SCREEN_HEIGHT)
about.minsize(SCREEN_WIDTH, SCREEN_HEIGHT)
about.resizable(0,0)

about_text = Label(about, text="TestGUI v1.0\n(C) Tom Simonsen, 2004")
about_text.pack(side="top")

about_button1 = Button(about, text="OK", command=about.withdraw)
about_button1.pack(side="bottom")

################################################################################
# Pop-ups:

# begin popup
popup1 = Toplevel()
popup1.withdraw()
popup1.title("About")
popup1.maxsize(SCREEN_WIDTH, SCREEN_HEIGHT)
```

```
popup1.minsize(SCREEN_WIDTH, SCREEN_HEIGHT)
popup1.resizable(0,0)

popup1_text = Label(popup1, text="The input has to be a number.")
popup1_text.pack(side="top")

popup1_button1 = Button(popup1, text="OK", command=popup1.withdraw)
popup1_button1.pack(side="bottom")
# end popup

root.mainloop()

# end gui
```

2.6.2 Windows CE programming

Windows CE is basically i scaled down version of the Windows operating system. It takes advantage of the Windows API, well-known classes and visual components and brings these to various hand-held device platforms. Windows CE was rebuilt entirely from the ground up with a completely new kernel source code that can run on a wide range of different CPUs. The footprint of Windows CE is very small compared to most operating systems for desktop systems, and it may only require as little as 1.5 megabytes of memory to run, depending on the version of the software. In this way, the Windows CE operating system is ideal for use with palmtop devices with limited resources and processing power. [Giannini and Keogh, 2001]

2.6.2.1 Windows CE versus other versions of Windows

There are a few issues that require attention when writing programs for Windows CE. Windows CE uses Unicode strings, meaning that characters are represented using 16 bits as opposed to the 8 bit ASCII character scheme that has become standard on many systems. The normal C string functions like `strcpy` or `strcat` will not work on Unicode strings, and functions like `wcscpy` and `wcscat` have to be used instead.

The displays for the hand-held devices are usually quite small with a relatively low screen resolution. Anything that is displayed on-screen has to be adapted so that it works well in such a small area. The display area of your program can be reduced further by panels that take up some of the screen, such as on-screen keyboards on devices with a touch sensitive display. The layout of the GUI should be constructed in such a way that these panels do not conceal important information.

Most Windows CE devices do not have a power-off button. These devices only offer a standby mode where the screen and sound is turned off to reduce power consumption. In other words, when a program is started, it may run for weeks or months. If a program has a memory leak error, this could prove to be fatal over such a long period of time, possibly leading to crashes or other malfunctions. Memory leaks are not released when you terminate the program in Windows CE, so a programmer has to be careful in maintaining the memory resources taken up by the program. Such memory leaks would have to be solved by a reboot of the device.

Windows CE has a hibernate mode to let an application go into hibernation to free up precious memory resources. When a program goes into hibernation it should release any unneeded memory so that it may be used by another applications. While in hibernation a program can receive a close or activate command. The close command should cause the program to close, and the activate should cause the program to acquire the resources that was released, if necessary, and start up where it left off before it was asked to hibernate.

When creating software for a palmtop device, the focus of attention should always be to design the code to use as little memory as possible. One should also favor this over speed. If a speedy data structure takes up too much memory, a slower one that takes up less memory should be chosen, for instance.

2.7 Compiler concepts

Compilers are programs that translate one language to another. The most common type of this sort of program is the type that translates a programming language to machine code. In this instance the *source language* is the programming language and the *target language* is the machine language. What I seek to do in this thesis is, as already mentioned, to translate one programming language to another programming language. Both the source language and target language will be programming languages. The same principles will apply to my compiler as to other compilers, regardless of the nature of the target language. Looking into the practices and principles of compiler construction before embarking on this mission of translating Python to C++ would probably be wise. This chapter seeks to briefly explain some of the compiler construction know-how that this thesis and the resulting software is based upon.

2.7.1 The anatomy of a compiler

There are many types of computer programs that are related to or used with compilers. These include interpreters, assemblers, linkers, loaders, preprocessors, editors, debuggers, profilers and project managers, to name a few [Louden, 1997]. I will not go into any of the details of these programs, nor is it my intention to produce any of the afore mentioned programs as a part of writing this thesis. I will look at what is traditionally known as the compiler, only.

A compiler consists of a number of internal *phases*. The phases perform distinct operations that may be coded separately or kept separately within the structure of the compiler itself. The phases of a compiler may include the *scanner*, the *parser*, the *semantic analyzer*, the *source code optimizer*, the *code generator* and the *target code optimizer* [Louden, 1997]. I will only focus on the parts of the compiler that are relevant to what I see as my task at hand, which are the scanner, the parser, the semantic analyzer and the code generator.

In addition to the various phases, the compiler also has a number of important data structures that the phases produce or rely upon. These are the *literal table*, the *symbol table* and *the syntax tree*. The literal table is used by the compiler to construct addresses for literals and for entering data definitions in the target code file. It is also used to reduce the amount of system memory that is taken up by the resulting program [Louden, 1997]. The literal table is really only necessary when the compiler outputs machine language, in my view. I will not look any further into the use of this data structure.

2.7.1.1 The scanner

The scanner is, as the name implies, the part of the compiler that scans the actual source code. It assembles the characters in the source code into meaningful entities called *tokens* or words. This process is known as *lexical analysis*. [Louden, 1997]

A token is usually represented symbolically in some way by the scanner. In some cases it is also necessary to store the original string of characters that make up the token in the source code. The names of variables and functions have to be kept for later use in the compiler, for instance. *Single symbol lookahead* is the most usual form of scanner algorithm, where the scanner only generates one token at a time. [Louden, 1997]

2.7.1.2 The parser

The parser relies on the results produced by the scanner. It receives the stream of tokens and performs a *syntax analysis* that determines the structure of the program. Syntax analysis is analogous to checking the grammar of the code, which is key to understanding the meaning of it. The parse phase of the compiler usually results in the construction of the syntax or parse tree data structure. [Louden, 1997]

2.7.1.3 The semantic analyzer

The *semantics* of the code is the meaning of it. This semantic analyzer checks the static semantics of the program. The semantics of a program determines the behavior of the program when it is running. The *dynamic semantics* of a program cannot be checked by a compiler, since the compiler does not actually run the program. The *static semantics*, however, are not altered by the program at runtime and can be checked. Mostly, this includes the checking of declarations and types of variables or expressions. The extra information produced by the semantic analyzer is called *attributes*, and these attributes are often used to “decorate” the syntax tree. This extra information can also be entered into other data structures used by the compiler. [Louden, 1997]

2.7.1.4 The code generator

The code generator is responsible for generating the target language [Louden, 1997]. It can do this based on the results from the source code optimizer or from the results of the semantic analyzer. In the latter case there is no optimization of the source code before the target language is generated. The target language is in most cases machine language, as mentioned above. In the case of this thesis the target language is not machine language, but another programming language, namely C++.

The target code can also go through an optimization phase, after it has been generated, to improve its efficiency. I have decided not to look into the optimization phases at all in this thesis.

2.7.1.5 The symbol table

This table structure (usually a hash table) interacts with almost every phase of the compiler program. An efficient data structure is of importance here, since this table is used very frequently. It stores information associated with identifiers, functions, variables, constants and data types. The symbol table does not have to be made up of one table, indeed it is not unusual to divide it into more than one table. A program could hold separate tables for variables and functions, for instance. In some cases the symbol table is not a table at all, but some other form of data structure that is suitable to serve its purpose. Such structures might be linear lists and various search tree structures, like binary search trees, AVL trees[1] and B trees[2], for instance. [Louden, 1997]

2.7.1.6 The syntax tree

The syntax tree is usually built by the parse phase of the compiler to represent the syntax of the source language and provide the storage for this syntax in the compiler's data structure. The syntax tree, in its most basic form, contains one node for each token that has been provided by the scanner. Usually, this basic syntax tree is turned into a simpler form, known as the abstract syntax tree (AST). The AST will still represent the exact same syntax, but with the use of fewer nodes. It is very common to remove the redundant nodes that represent certain types of brackets and encapsulations, for instance.

2.7.2 Finite automata

Finite automata are closely related to the scanning phase of a compiler. They are a mathematical way of describing particular kinds of algorithms and can be used to recognize string patterns. Thus a scanner algorithm can be constructed from finite automata. Sometimes referred to as finite-state machines, finite automata are related to regular expressions. [Louden, 1997]

2.7.2.1 Deterministic finite automata

In deterministic finite automata (DFAs) the next state of the machine is uniquely given by the current state of the machine and the character that is currently being scanned. There is a strong relationship between a regular expression for a string pattern and a DFA that *accepts* strings according to it. A DFA has accepted a string when it has reached one of its final states. The *language* accepted by the machine is the set of strings it accepts. [Lewis and Papadimitriou, 1998]

2.7.2.2 Nondeterministic finite automata

Nondeterminism is basically the ability to change states in a way that is only partially determined by the current state and the current input character. Where there is only one possible next state in a DFA there might be many possible next states to chose from in nondeterministic finite automata, or NFAs. The machine can chose to skip to any of the legal states as it is scanning, and this changing of states is not defined by any rule in the model. It is nondeterministic in the way that there is no rule for which of the next legal states the machine is going to chose. An NFA is equivalent to a DFA in the sense that they will accept the same language. An NFA will usually have fewer states than a DFA, but it will still be able to do the same job. [Lewis and Papadimitriou, 1998]

2.7.3 Context-free grammars

A *context-free grammar* specifies the syntactic structure of a programming language using recursive rules. A context-free grammar uses naming conventions and operations that look similar to those of

1 An *AVL tree* is a balanced binary tree where the height of the two subtrees of a node differs by at most one.

2 A *B tree* is a balanced search tree in which every node has between $\lceil m/2 \rceil$ and m children where m>1 is a fixed integer. m is the order. The root may have as few as two children.

regular expressions. The rules of the context-free grammar are recursive, however, to allow for the nesting of various syntactic elements inside of each other. An if-statement can contain other if-statements and a while-loop can contain other while-loops, for instance. Since the notation for the grammar specification was developed by John Backus and Peter Naur, such grammar rules are said to be in *Backus-Naur* form, or BNF. [Louden, 1997]

2.7.4 Parser algorithms

A *parser algorithm* has the task of determining the syntax of a program. It does the syntax analysis based on the grammar rules, usually given by a context-free grammar. The parsing algorithms differ a great deal from the scanning algorithms. They usually depend on recursive calls and larger data structures. Where the scanning algorithm does not depend on much data structuring, the parsing algorithm usually maintains a *parsing stack* to help it build the syntax tree. The tree structure is representing the recursive nature of the syntactic structure. A linear structuring of the data would not be able to do this. There exists a number of different parsing algorithms that mainly falls into two major categories: *top-down* and *bottom-up parsing*. [Louden, 1997] I will not go into further details surrounding the algorithms. I will simply point out some of the various algorithms that exist. Plenty of good literature can be found on this subject.

2.7.4.1 Top-down parsing

A top-down parsing algorithm creates the parse or syntax tree from the root node to the leaf nodes. The traversal order of the resulting syntax tree is preorder. There are two forms of top-down parsing algorithms: *backtracking parsers* and *predictive parsers*. Some predictive parsing algorithms are *recursive-decent*, *LL(1)*, *LL(k)* and *SLL(k) parsing*, to name a few. A kind of backtracking parsing algorithm is called *combinator parsing*. [Louden, 1997]

2.7.4.2 Bottom-up parsing

A bottom-up parser builds the syntax tree starting at the leaf nods and works its way to the root node. The most general bottom-up parsing algorithm is called *LR(1)*[1] parsing. Variations on this basic algorithm includes *LR(0)*, *SLR(1)*, and *LALR(1)*[2] *parsing*. The Yacc parser generator software (described in chapter 3.2.1.2) generates LALR(1) parsers.

2.8 The tools

I have used a number of different software packages to develop the code for the Python to C++ translator. The Python source code has been written in Python version 2.4, using the IDLE version 1.1 from the Python Software Foundation. The handwritten C and C++ source code has been developed using Emacs version 21.3.1 and compiled using both g++ (on the Linux platform) and Open Watcom version 1.3 (on the Windows XP platform). The translated code for Windows CE has been assembled and compiled in Microsoft's eMbedded Visual C++ version 3.0.

1 The "L" in LR(1) indicates that the input is processed from left to right, the "R" indicates that a rightmost derivation is used and 1 indicates that one symbol of lookahead is used.

2 LALR(1) stands for *lookahead LR(1)* parsing. A complete definition of this type of algorithm is found in Kenneth C. Louden's book on compiler construction [Louden, 1997], among others.

Code that has been translated to “standard” C++ has been tested on the Windows XP and Fedora Core 3 Linux operating systems. Windows CE GUI code has only been tested on the Hewlett-Packard iPAQ device running the Microsoft Pocket PC 2002 operating system.

3 Many ways to solve a problem

3.1 Using regular expressions

Regular expressions represent patterns of strings of characters. A regular expression is completely defined by the set of strings that it matches. [Louden, 1997] This set is called the *language generated by the regular expression*. The word *language* in this context is used to mean “set of strings”. Python has the support for regular expressions built into it. This is a very powerful string recognition tool. A regular expression in Python will recognize any sequence of characters in any given string or text.

Since Python's regular expressions can be used to recognize strings as they occur in a text, this tool could be useful for building a translator program. The regular expressions would be able to recognize the various string elements that make up the source code of a program. The basic idea is that a translation program that is built using regular expressions has a very simple structure. It does not rely on large data structures like parsing stacks, syntax trees or variable tables to be able to translate one language to another. Nor does it rely on having typical scanner and parser algorithms to be able to read the code and decipher its meaning. The idea is that the string patterns that are matched can be turned into the equivalent string patterns in the target language, on the fly. The while loop in Python, for instance, will make up a set of certain string patterns that can be recognized by one or more regular expressions. All of these Python string patterns will have their equivalent C++ string patterns. It is just a matter of matching them up.

Joe Strout and Dirk Heise's “Python2C” is a short and simple attempt at making a Python program that translates Python code to C/C++ code [Strout and Heise, 1998]. This attempt at creating a Python to C translator also inspired me to give the same name to my own attempt at translating Python to C++. The very first investigations I made in connection with this thesis were related to what Strout and Heise where trying to do with their Python2C. It is a small experiment into what can be done with regular expressions alone. The source code for Strout and Heise's Python2C program is available on Joe Strout's website. By making some additions to Strout and Heise's program, I set up a modified version of this code. The complete source code for this modified version is included in the appendix.

3.1.1 Strout and Heise's Python2C

The translator code in Strout and Heise's Python2C mainly consists of regular expressions and provides support for translation of a few of the basic elements of the Python programming language to C. This includes loops, variable assignment and other basic utilities. The translator also has some support for some types of C++ statements.

The Python2C code does not comprise many lines of code, and yet it seems to manage to translate many of the basic features of Python to C/C++. The program is only meant do be a small experiment, and it will just translate a small subset of all possible Python statements to C/C++.

Translating Python to C++ for palmtop software development - 3 Many ways to solve a problem

The following example has been taken from the web page of Joe Strout and can be found at the following web address: http://www.strout.net/python/ai/python2c_demo.html

This Python code was fed to python2c.py:

```
# this is a test
x = -1
while x:        # loop until 0 spam
        x = input('\nHow much spam?')
        if x:
               print "We have:"
               for i in range(0,x):
                       print 'spam',
                       if i == x-2:   # (don't forget the beans!)
                               print "baked beans and",
        else:
               print "Enjoy!"
# all done!
```

...which spewed out the following C++ code in response:

```
// this is a test
x = -1;
while (x)   // loop until 0 spam
{
    cout << "\nHow much spam?"; cin >> x ;
    if (x)
    {
        cout << "We have:" << endl;
        for (i=0; i<x; i++) {
        {
            cout << "spam" << " ";
            if (i == x-2)   // (don't forget the beans!)
            {
                cout << "baked beans and" << " ";
            }
        }
    }
    else
    {
        cout << "Enjoy!" << endl;
    }
}
// all done!
```

3.1.2 Problems with the regular expressions approach

As seen in the execution example take from Joe Strout's website, Strout and Heise's Python2C does not handle the problem concerning variable creation. My amended version of the program does not do anything to correct this problem either. To make the resulting C++ code from the example run through a compiler without errors, one would have to edit the code by hand. The variable `x` in the translated C++ code is initialized but never declared. The same is true for the variable `i`. The type of the variable is never explicitly declared in Python, as it must be in C/C++. To declare a variable in

C/C++ the translator has to find out what type of value variable `x` stores. The only good solution to this problem, the way I see it, is to scan and parse the code just like a compiler would. When you have a statement such as "`x = -1;`", the problem of finding out what type `x` holds is trivial. If the statement is something like "`x = class.function(y)[1]`", it is a whole different matter. A statement such as this can be picked apart by a traditional parser algorithm very efficiently. When the parser has found the meaning of what is on the left of the "=", the type of `x` has been found in the process. How the same thing can be achieved through the use of regular expressions is uncertain. It is unclear how the matching of string patterns in this way will acquire the meaning of code.

The regular expressions approach runs into further problems, relating to the structuring of the code. The approach gives rise to a number of questions. When the translator program comes across some variable, how does the translator program know that it has not encountered the same variable before? If it has encountered a variable with the same name before, what was the scope of that variable? When encountering a function call, is there any way of determining the return type of the function?

When dealing with function calls that call functions that are defined in the Python language, the return type problem can be solved by having look-up tables. If a function is not defined in the Python language, how can the return type of a function be determined when the function's definition or code has to be translated? If you are able to determine the return type, how do you store it in your translator program? How does a regular expression translator program know if a function belongs to a class or not? How does such a translator program know that a piece of code belongs to a function, a while loop, a for loop or an if statement? The questions go on.

The problem with the regular expression approach is that the matching of the strings is done with no regard for the context in which the strings are found. This approach can probably work fairly well on short and simple programs, but most likely not on larger programs with complex data structures. As soon as we start to talk about variable type, scope, functions and classes, we will run into trouble. In essence, the regular expression translator program is an implementation of some sort of a scanner algorithm. It has all the functionality required to make it a scanner, but all to little of what is required to make it a parser.

There are other problems related to the regular expression approach found in Strout and Heise's Python2C, and one of them relates to the balancing of parenthesis. If you have a string containing multiple parenthesis like this example; "`((()())((())()))()`", it is quite difficult to write a regular expression that would pick out the pairs of left and right parenthesis that belong to each other. If you use a standard scanner algorithm however, that reads one character at a time, it is quite easy to pick these pairs out. The "balancing of parenthesis problem" crops up more than once when translating code. Almost every code element can be enclosed in parenthesis and various forms of brackets. Consider the "`{`"and the "`}`"in C/C++, for instance. This turns out to be a big problem for a translator using regular expressions in the way that it is being used in Strout and Heise's Python2C.

I feel that it is beyond the scope of this thesis to try and find answers to the many questions and problems above. My conclusion is that I will not pursue the "pure" regular expression approach to building a Python to C++ translator. There are just too many unanswered questions. I can speculate that a hybrid solution might do the job properly. Such a hybrid solution could be a solution where the code is scanned and possibly also parsed by the regular expressions and the meaning of the code is represented by some sort of a syntax tree data structure. This solution would perhaps look something like the solution provided by Lex and YACC, described in chapter 3.2.1.

3.2 Creating a scanner and parser from scratch

3.2.1 Using Lex and Yacc

3.2.1.1 Lex

Lex is a popular scanner generator program that exists in a number of different versions. The most popular version is called “flex”, for *Fast Lex*. This version is produced by the Free Software Foundation and is freely available at many websites [Louden, 1997]. Lex can generate a scanner program for any language from a description of the tokens of the language. The description for each token must be a regular expression. The Lex conventions for writing regular expressions do not differ much from the conventions found in the Python language. I will not go into the details of these in Lex or Python. Instead I will provide an example of a Lex scanner that I wrote for a small and simple language called *Pedagola*. The source code is included in the appendix.

Lex takes a text file as input that needs to contain the regular expression descriptions paired with the actions to be taken when each expression is matched. Lex outputs a file, usually named `lex.yy.c` or `lexyy.c`, that contains the C source code for the scanner program. This file can than be compiled and linked to a main program to get a running piece of software. The C source code defines a procedure called `yylex` that is a table-driven implementation of a DFA corresponding to the regular expressions in the input file. [Louden, 1997]

3.2.1.2 Yacc

Yacc, which stands for *Yet Another Compiler-Compiler*, is a parser generator program. Historically, parser generators were called compiler-compilers. Yacc takes a specification of a language's syntax as input and produces a parse procedure for the language as output. This procedure is an implementation of a LALR(1) (See chapter 2.7.4.2) parsing algorithm. There are a number of different versions of Yacc to be found, also several public domain versions under the name of *Bison*. One popular version is the GNU Bison, distributed by the Free Software Foundation. [Louden, 1997]

The Yacc output file consists of C source code. This file is usually named `y.tab.c` or `ytab.c`. The input or specification file has a basic format that divides the file contents into three sections. The sections are the definitions section, the rules section and the auxiliary routines section. These are separated by lines of double percent signs. [Louden, 1997]

The definitions section contains the information about the tokens, data types and grammar rules that Yacc needs to build the parser. It also includes any C code that must go directly into the beginning of the output file. These are often `#include` directives of source code files that make up the other parts of the parser program. [Louden, 1997]

The rules section contains grammar rules for the language in a modified BNF form [Louden, 1997]. For each rule there are actions in C code that are to be executed every time the associated grammar rule is recognized. I have included an example of a Yacc input file for the Pedagola language in the appendix, together with the Lex input file for the same language. This will exemplify what a scanner and parser will typically look like when using the Lex and Yacc software tools, and how much work that has to be invested into making them.

The third and last section of the Yacc input file makes up the auxiliary routines. It contains procedures and function declarations that may not be available through `#include` directives and that are needed to complete the program. [Louden, 1997]

3.2.1.3 Using Lex and Yacc to create my translator program

Creating a Python scanner and parser from scratch using Lex and Yacc would involve a lot of work. When looking at what it takes to get the compiler for the small and simple Pedagola language up and running, one has to realize that it is not realistic for one man to do the same thing - and then some - to create a Python translator in the time set aside for this thesis. The Pedagola compiler code in the previous chapter is far from complete, since it only checks the syntax and the grammar of a Pedagola program. It does not even check all grammar rules. It does neither compile or translate anything, and no machine language is generated. A lot more code would have to be added for that to be possible.

The Python BNF is a lot larger and more complex than the short and simple Pedagola BNF. A lot of work and too much time would have to go into getting a correct syntax tree generated by the software. Creating the syntax tree is not what this thesis is really about, and the workload put into the development process should be focused around the code that does the actual translation. I want to look at what can be generated from this syntax tree in terms of translated code. It would be preferable if there exists some tool that could provide me with a correct syntax tree in an easier and faster way.

3.2.2 Really creating a scanner and parser from scratch

Based on my final notes regarding using the Lex and Yacc software tools in the previous chapter, actually writing a scanner and a compiler for the Python language, entirely from scratch, is a monstrous task. No tools would be used to generate anything and every line of code would have to be written by hand. It is completely unrealistic for one programmer to do this in the space of such limited time. Thus, this way of solving the problem should be dismissed all together.

3.3 Using existing modules in Python

The Python `compiler` and `parser` modules provide functions to access the Python internal compiler and parser. The `compiler` module contains functions to parse and compile the contents of files using the Python compiler. A module called `compiler.visitor` contains a function to walk the Python AST, called `walk`. The `compiler.ast` module contains further functions to get at the nodes stored in the AST. The Python parser and byte-code compiler can be accessed through the functions found in the `parser` module. This is a module that has been created for the purpose of being able to access and manipulate the AST and its objects.

By using the functions found in the `compiler`, `compiler.ast` and `compiler.visitor` modules I am actually able to create, access and traverse the Python AST without having to create a compiler from scratch. I can also access each AST node and its contents through the `walk` function The programmers at Python.org as tried and tested this code and I can be fairly certain that the AST contains a minimum of bugs. These modules provide me with the tools to create a syntax tree without much effort and to be able to focus the workload on the translator code.

3.3.1 Using the Python modules in practice

To create the Python AST the Python compiler must parse the Python source code in question. The Python source code to be translated will be found in files, in my case, and the contents of files can be parsed by using the `parseFile` function found in the `compiler` module. This function returns a pointer to the Python AST structure. This pointer can be passed as an argument to the `walk` function, found in the `compiler.visitor` module.

To be able to get the `walk` function to actually do its job a *visitor function* for each type of AST node has to be implemented in the translator program's code. In the example below all the visitor functions have been put inside a class. An instance of this class is passed as the second argument to the `walk` function. The name of each visitor function starts with “`visit`” followed by the type of node it is programmed to visit. If the type of the node in question is “`Add`”, then the name of the corresponding visitor function is “`visitAdd`”. The `walk` function will detect the type of each AST node and call the appropriate function accordingly. If the corresponding visitor function is not found, a default function is called and the subtree of the node will not be traversed. Totally, there exists some sixty or so types of AST nodes. If the translator program is going to be able to traverse any Python AST, then all of these types of nodes must have a corresponding visitor function.

For the subtree rooted at each AST node to be traversed, the `walk` function has to be called inside every visitor function, as indicated by the example below. A child node is passed as the first argument to the `walk` function and the parent node or root node of the subtree is passed as the second. The children of an AST node can be accessed though the `getChildren` or `getChildNodes` functions found in the `compiler.ast` module. The `getChildren` function returns a list of AST nodes and other objects stored in the parent AST node, and the `getChildNodes` function returns only a list of the AST nodes. As indicated by the example below, it does not take many lines of code to get this to start working.

This short Python code example shows how using the Python modules to traverse the Python AST could work in practice. “. . . ” indicates that the code is incomplete and that more lines of code is needed.

```
import compiler
from compiler.ast import *
from compiler.visitor import *

class VisitorFunctions:
    def visitAdd(self, node):
        ...
        children = node.getChildNodes()
        for child in children:
            walk(child, self)
        ...
    def visitAnd(self, node):
        ...
        children = node.getChildNodes()
        for child in children:
            walk(child, self)
        ...
    ...

# Parse the Python source code:
ast = compiler.parseFile(python_file_name)

visitor_functions = VisitorFunctions()

# Walk the Python AST:
walk(ast, visitor_functions)
```

3.4 The hybrid

The hybrid solution to creating a Python to C++ translator is a solution that may borrow from all the above ways to solve the problems that are encountered. I think I have pointed out the need for a scanner/parser algorithm quite clearly in the previous chapters. To be able to translate code well, the translator program needs to know what the code means. This can only be done by checking the grammar and semantics. The most obvious and tried and tested solution is to use a standard scanner algorithm to scan the code and a standard parser algorithm to build a syntax tree to represent the meaning of the code. When looking at Strout and Heise's Python2C in chapter 3.1.1, I concluded that their attempt at creating a translator, based on regular expressions, would not work. The segments of code are translated out of context and the meaning of each segment is never picked up by the program. Strout and Heise's Python2C is only a scanner, although an incomplete one. The other solutions I have looked at have got the scanner and the parser algorithms built into them. One "hybrid solution" may be the use of regular expressions to create a translator program that incorporates a scanner and a parser algorithm.

Regular expressions can be used in various parts of the program. The most obvious use would be to use regular expressions as a basis for the scanner algorithm, quite similar to the way that the regular expressions are used by Lex. It could also be used as a tool that picks out certain parts of the target code for code optimization. Here it might look something like Strout and Heise's Python2C, and it does not have to recognize one token at a time. This sort of code optimization would be based on recognizing code patterns that have an inefficient form. Many such patterns are relatively independent of the context or the meaning of the code. Regular expressions can also be used rather efficiently to determine the type of constants in Python code, as is shown by the example below.

The below Python code shows how the types of constants can be determined using regular expressions. This example requires the Python `re` module to be imported.

```
class LangConst(LanguageConstruct):

    parent = None
    value  = ""
    type   = ""

    def __init__(self, p, v):
        self.debugInfo()
        self.parent = p

        self.value = str(v)
        #The type can be determined based on the contents of the "value" string.
        integer_pattern       = r"[+\-]?\d+"
        hex_pattern           = r"0x[01]+"
        scientific_pattern    = r"[+\-]?\d(\.d+|)[eE][+\-]\d\d?"
        decimal_pattern       = r"[+\-]?(\d+\.\d*|\d*\.\d+)"
        float_pattern = r"(" + scientific_pattern + "|" + decimal_pattern + r")"

        match = re.match(hex_pattern, self.value)
        if(match != None):
            self.type = "HEX"
        else:
            match = re.match(float_pattern, self.value)
            if(match != None):
                self.type = "FLOAT"
            else:
                match = re.match(integer_pattern, self.value)
                if(match != None):
                    self.type = "INT"
                else:
                    self.type = "STRING"
```

3.5 Other attempts

3.5.1 Ariel

Ariel is a research project to investigate the design of user interfaces that “take advantage of natural means of communication such as speech, gestures, and facial expressions” [Hugunin, June 1996].

Here, a working prototype was constructed using Python. From this initial version of the software it was evident that parts of the Python code were acting like bottlenecks in the program, and needed to be sped up somehow. The solution was to translate these slowly executing segments of Python code into C. After having translated a bit of C code it became evident that much of the translation was quite trivial in places, and an attempt was made to automate this process.[Hugunin, June 1996] Unfortunately, I have not come across any samples of the code or any code documentation from this project.

3.5.2 Pyrex

"Pyrex is a language specially designed for writing Python extension modules" [Ewing, 2002], since writing these extensions can be quite messy using C/C++.

Pyrex has turned out to use most of the usual Python syntax, with some amendments like the introduction of having to declare types. As mentioned earlier, types are not declared in Python, and a variable can be of several types during execution. In Pyrex you have to add the word “cdef” ahead of every variable declaration. A typical such statement in Pyrex would look something like: “`cdef int x`”.

Since the developers of Pyrex basically have created a new language based on the Python language, this project does not attempt to solve the Python to C++ translation problem as such. The ambitions of the Pyrex developers were first and foremost to develop an easy-to-use translator that had the abilities to produce fairly efficient C code. It did not matter whether this meant that changes had to be made to Python itself. In this way the development of Pyrex differs from what I wish to do. My intentions are to create a translator that can translate pure Python to C++, without the use of amendments such as the ones found in Pyrex.

3.6 Conclusion

I have gained some knowledge from my encounters with Lex and Yacc and Strout and Heise's Python2C. The hands-on approach with Lex and Yacc gave me some more insights into the inner workings of a compiler, and the scanner and parser algorithms. I used the hands-on approach with Strout and Heise's Python2C as well, and found many weaknesses in using just regular expressions to create a translator.

As I have mentioned earlier, I do not want to spend a lot of time creating a Python compiler. Since I wish to focus my attention on the code translation process, I want to be able to build and get access to a syntax tree in the easiest possible way. Creating a compiler entirely from scratch or using Lex and Yacc to do this is not an option. It would take a lot of time to create a complete and working Python compiler. The best solution available, as I see it, is to use the existing modules in the Python language to build and access the Python syntax tree. I have already shown how this can be done in chapter 3.3.1, using just a few lines of code.

4 My Python2C

I have chosen to create a Python to C++ translator that will use the existing Python modules to build and access the Python AST. As a tribute to Joe Strout and Dirk Heise, I have named it “Python2C”.

In this chapter I will start with the code documentation and go on to describe and discuss the various translation problems that I have encountered. The code documentation will provide the overview of the inner workings of my Python2C translator program. When the inner workings and major data structures have been pointed out, I go on to describe and discus the translation problems.

4.1 Source code documentation

4.1.1 The main code

The main or global part of the code ties it all together. This is the Python code that does not reside inside of any class structure in Python2C, and it is there to provide the simple textual user interface and the functions that start the translation process.

4.1.1.1 The user interface

Through the textual user interface the user can set a number of global variables that determine the output of Python2C. The list of the commands or options that are available through Python2C's command line is displayed when typing in the “-help” option, followed by pressing the “enter” key. This list is included as it appears in the Python2C command line, below. No options can be sent to Python2C directly from the command line of the operating system. Python2C does not check the input to the program via the `sys.argv` list, and the program has to be started before anything can be passed to it.

This is a snapshot of Python2C displaying the short start-up information and the help menu as it appears in Python2C version 0.014.

```
### Python2C version 0.014
### Programmed by Tom Simonsen (tom.simonsen@gmail.com).
###
### Type in a file name or various options below.
### Type "-help" to display options.
### ->: -help
###
### ##########################################################################
### #                                                                        #
### # Python2C options:                                                      #
### #                                                                        #
### # file_name          (Parse and translate file with name file_name.)    #
### # -p                 (Turn on/off only parsing mode.)                   #
### # -p file_name       (Turn on/off only parsing mode.)                   #
### # -g                 (Turn on/off graphical mode.)                      #
### # -g file_name       (Turn on/off graphical mode.)                      #
### # -s path            (Set the source code folder.)                      #
### # -d path            (Set the destination folder.)                      #
### # -salloc number     (Set the string allocation size.)                  #
### # -tstnodes          (Turn on/off the printing of TST-nodes.)           #
### # -tstnodes file_name (Turn on/off the printing of TST-nodes.)          #
### # -vtable            (Turn on/off the printing of the variable table.) #
### # -vtable file_name  (Turn on/off the printing of the variable table.) #
### # -ftable            (Turn on/off the printing of the function table.) #
### # -ftable file_name  (Turn on/off the printing of the function table.) #
### # -ptree             (Turn on/off printing of the parser tree.)         #
### # -ptree file_name   (Turn on/off printing of the parser tree.)         #
### # -q                 (Quit Python2C.)                                   #
### # -help              (Display this menu.)                               #
### #                                                                        #
### ##########################################################################
###
### ->:
```

Many of the commands that are available in Python2C can be given in two ways. Either they can be entered followed by a string, or only the command itself can be entered. The “two-way” commands are “-p”, “-g”, “-tstnodes”, “-vtable”, “-ftable” and “-ptree”. If you enter “-p” the “only parsing mode” is activated. If you enter “-p” followed by the file name “hw.py”, the only parsing mode is activated and the file named “hw.py” is parsed by Python2C. You can set more than one mode by entering the various commands in any order. Modes “-p” and “-g”, for instance, can be activated at the same time. “-p” can be entered – that is; you have to press the “enter” key after typing it - followed by “-g” and the “enter” key. This will sett both the parsing mode and the graphical mode. If you now enter a filename, this file will be processed with these two modes activated.

Entering the same command a second time will deactivate the mode. By default none of the Python2C modes are activated upon start-up, unless you make changes to the values of some of the global variables that govern these modes, in the source code.

The simplest way to get something done is to just enter the name of a file into Python2C. This will cause Python2C to translate this file, if it exists. If only the name of a file is entered into the program, then Python2C will assume that this file resides in the current folder, which is normally the same folder as where Python2C's source code is found. If this is not the case, the *source folder* has to be set. The source folder is set with the command "-s" followed by the path to the specific folder (for instance "-s /python/source/"). The "-s" command has to be followed by a valid path or an error message will be displayed. When a valid path has been entered, Python2C will assume that all the file names that are entered refer to files in this folder. The source folder can of course be changed at any time. The `source_folder` variable holds the source folder path string in the source code.

The user can also set and change the path to the *output folder* or *destination folder* of Python2C. By default, this is also the current folder. All the resulting files of the translation are created or overwritten in the destination folder. (More information on all the files produced by Python2C is to follow in this chapter.) The destination folder is basically set in the same way as the source folder. The difference is that you must enter "-d", for destination, followed by a valid path to the desired folder. Until the path is changed, all the translation data will end up in the same folder. Python2C does not ask before overwriting the files it produces, so some care has to be taken to ensure that this does not produce undesired results. Python2C generates a substantial number of files, so decided it would probably be inconvenient to make Python2C ask the user for permission to overwrite already existing files. The names of the generated files are a bit unusual, so it is unlikely that they will overwrite files that have nothing to do with the Python2C translation. The `dest_folder` variable holds the destination folder path string.

The "-p" command turns the *only parsing mode* on or off. The only parsing mode will make Python2C do basically everything it normally does, except for outputting the translated C++ code. The code is still parsed by the the Python compiler module, the AST is still traversed and the TST (the Translated Syntax Tree. See chapter 4.1.8.) is still created. All the debugging information and data files are produced as well. This mode exists mostly for debugging purposes. The Python2C execution can be seen as having two major stages: the "parsing stage" and the "translation stage", named very loosely. For debugging purposes it can be very useful to be able to separate these two stages and deal with them separately. If errors are produced in the first stage more errors are certain to occur in the second, since the second stage relies on all the data created in the first stage. Stage one, which I have named the "parsing stage", creates a fair deal of output in the form of data files. All of these can be checked to verify that the program is functioning as desired when given a specific set of input data. All of this can be done before Python2C has done any actual code translation. When one has made sure that the parsing stage is setting up the TST in a correct manner, the second stage can be activated. If any errors should occur now, one can be relatively certain that they are originating from within the code that makes up the second stage, the "translation stage". Most of the actual translation work is already done in the first stage, in the process of setting up the TST. The actual creation and output of the C++ source code is done in stage two. This stage relies heavily on the parsing stage having been able to produce all the required data and having put it in the TST correctly. If this is not the case, Python2C will usually crash. The parsing mode is set by the `translation_mode` variable in the Python2C source code. Its default value is 1, where it makes the program go through with the translation to C++. Set it to 0 to activate the only parsing mode.

The "-g" command turns on or off the graphical mode of translation. This mode assumes that the Python file that is going be translated has a graphical user interface that needs to be translated. The Python GUI code will be translated to Windows CE GUI. This mode could also be set automatically by Python2C, instead of manually by the user, when it finds that the `Tkinter` module is imported in the code that it is translating. Currently, I have chosen not to implement this. This would require the `Tkinter` module to be imported only at the beginning of the source code file. Nor has any actual support for GUI code translation been implemented (See chapter 4.2.4.). The graphical mode is set by the `graphical_mode` variable in the Python2C source code.

Every Python string variable is translated to a character pointer in C++. All character pointers have to have some memory set aside for them by allocating memory through the `malloc` function in C++. The "-salloc" command sets the allocated size of all the C++ character pointers that are produced by the translation. "-salloc" has to be entered followed by an integer. The default value of this integer is 256 and is stored in the `malloc_string_size` variable.

An example showing how to activate and deactivate modes in Python2C.

```
### Python2C version 0.012
### Programmed by Tom Simonsen (tom.simonsen@gmail.com).
###
### Type in a file name or various options below.
### Type "-help" to display options.
### ->: -p
### Only parsing mode activated.
### ->: -tstnodes
### Printing of the TST-nodes activated.
### ->: -p
### Only parsing mode deactivated.
### ->: -q
### Exiting Python2C.
```

The "-tstnodes" command activates printing of the TST nodes as they are created by Python2C. Only the name of the class of the node is being printed for each node in addition to a number indicating how many nodes have been created so far. Some of these nodes are intermediate and do not end up as being a part of the TST data structure. Never the less, the printing of these nodes give you an overview of the nodes that end up being created and roughly what the TST ends up looking like. This information on the TST nodes are not printed to file as with some of the other debugging data output by Python2C. All the nodes are printed out on the command line as the the parsing stage of the program is running. By default the printing of the TST nodes is turned off. The variable controlling this feature is named `print_tst_nodes` in the Python2C source code.

Below is an example of what the "-tstnodes" mode looks like, taken from the command line.

```
### Python2C version 0.012
### Programmed by Tom Simonsen (tom.simonsen@gmail.com).
###
### Type in a file name or various options below.
### Type "-help" to display options.
### ->: -tstnodes hw.py
### Printing of the TST-nodes activated.
### Parsing and translating file "..\Tests\hw\hw.py".
1 LangModule
2 LangStmt
```

```
3 LangMain
4 LangImport
5 LangAssign
6 LangAssName
7 LangAssName
8 LangCallFunc
9 LangName
10 LangSubscript
11 LangGetattr
12 LangName
13 LangConst
14 LangCreate
15 LangCreate
16 LangAssign
17 LangAssName
18 LangAdd
19 LangConst
20 LangCallFunc
21 LangGetattr
22 LangName
23 LangName
24 LangCreate
25 LangAssign
26 LangAssName
27 LangConst
28 LangCreate
29 LanguageConstruct
30 LangPrintnl
31 LangAdd
32 LangAdd
33 LangAdd
34 LangConst
35 LangCallFunc
36 LangName
37 LangName
38 LangConst
39 LangCallFunc
40 LangName
41 LangName
### Debug info files written.
### Additional source files written.
### Code translated and written to files.
###
### About the execution:
### Nodes in the AST: 35
### Total number of TST nodes created (All may not be part of the TST
structure.): 41
### Parsing of the AST and creation of the TST completed in 0.4276 seconds.
### Translation completed in 0.0007 seconds.
###
### ->:
```

The activation of the “-vtable” command will output the same information that is found in the “`*_var_stack.txt`” data file. “*” represents the name of the file containing the Python code to be translated, without the file name extension. The content of the variable table is printed to the command line for each new addition that is made to it. By default this option is turned off in Python2C. The variable that governs this mode is the `print_var_table`.

Below is an example of a Python2C execution with the "-vtable" option activated.

```
### Python2C version 0.012
### Programmed by Tom Simonsen (tom.simonsen@gmail.com).
###
### Type in a file name or various options below.
### Type "-help" to display options.
### ->: -vtable hw.py
### Printing of the variable table activated.
### Parsing and translating file "..\Tests\hw\hw.py".
VariableTable
<Global level> {"r" : LangAssName <type FLOAT>, }

VariableTable
<Global level> {"r" : LangAssName <type FLOAT>, "t" : LangAssName <type
FLOAT>, }

VariableTable
<Global level> {"s" : LangAssName <type FLOAT>, "r" : LangAssName <type FLOAT>,
"t" : LangAssName <type FLOAT>, }

VariableTable
<Global level> {"s" : LangAssName <type FLOAT>, "r" : LangAssName <type FLOAT>,
"u" : LangAssName <type STRING>, "t" : LangAssName <type FLOAT>, }

### Debug info files written.
### Additional source files written.
### Code translated and written to files.
###
### About the execution:
### Nodes in the AST: 35
### Total number of TST nodes created (All may not be part of the TST
structure.): 41
### Parsing of the AST and creation of the TST completed in 0.1437 seconds.
### Translation completed in 0.0014 seconds.
###
### ->:
```

The "-ftable" command activates the output of the function table in the same way as the "-vtable" command activates the output of the variable table. This is also the same information that is found in one of the debugging information files, namely the "`*_func_table.txt`" file. When the "-ftable" option is active, the content of the function table is printed to the command line each time a new function is added to this table. In the Python2C source code the variable `print_func_table` governs the "-ftable" printing mode.

I have included an example of the translation of a file named "`test08_functions.py`" below. In the example you can see that the return type of the function `printOut` has not yet been determined when the function is added to the function table. This is a problem that occurs fairly frequently. Often, the statements that determine the return type of the function are located further down the code than the definition of the function itself, leading the return type of a function to be determined at some later point in the parsing stage. If you take a look at the translated C++ code (also included below), the return type has indeed been determined at the point of translation. The return type in this case is "`void`".

Below is the Python source code found in the file "`test08_functions.py`".

```
#!/usr/bin/env python

def printOut(number, string):
    s = ""
    for i in range(number):
        if(i % 5) == 0:
            print "\n" + string
        else:
            print string

string = raw_input("Type a string: ")
number = int(raw_input("and the number of times it should be printed: "))
printOut(number, string)
print "\nString \"" + string + "\" printed " + str(number) + " times."
```

The execution example below shows what is printed to the command line when running the `test08_functions.py` code through Python2C with the "-ftable" option activated. As you can see, the return type of the `printOut` function has not been determined at the point of the function's addition to the function table.

```
### Python2C version 0.012
### Programmed by Tom Simonsen (tom.simonsen@gmail.com).
###
### Type in a file name or various options below.
### Type "-help" to display options.
### ->: -ftable test08_functions.py
### Printing of the funciton table activated.
### Parsing and translating file "..\Tests\test08\test08_functions.py".
<"Global level"> {"printOut" : LangFunction <return type = >, }

### Debug info files written.
### Additional source files written.
### Code translated and written to files.
###
### About the execution:
### Nodes in the AST: 56
### Total number of TST nodes created (All may not be part of the TST
structure.): 79
### Parsing of the AST and creation of the TST completed in 0.1343 seconds.
### Translation completed in 0.0014 seconds.
###
### ->:
```

Finally, the code below shows the translated C++ source code. Here, the return type of the `printOut` function has been determined.

```
/* File translated Tue Jun 07 15:55:41 2005
 * by Python2C version 0.012. */
```

```
#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>

/* The python types translation headers: */
#include "python2clist.h"
#include "python2ctuple.h"
#include "python2cdict.h"

/* Forward declarations: */
void printOut(int number, char* string);

void printOut(int number, char* string)
{
  char* s;
  int i;
  s = (char*) malloc(256);
  s = "";
  for(i = 0; i < number; i += 1)
    {
      if(i % 5 == 0)
      {
        cout << "\n" << string << endl;
      }
      else
      {
        cout << string << endl;
      }
    }
}

char* string;
int number;

int main(int argc, char* argv[])
{
  printf("Type a string: ");
  string = (char*) malloc(256);
  scanf("%s", string);
  printf("and the number of times it should be printed: ");
  number = atoi(scanf("%s"));
  printOut(number, string);
  cout << "\nString \"" << string << "\" printed " << number << " times." <<
endl;
  return 0;
}
```

The “-ptree” command activates or deactivates the printing of the Python parser tree as it exists in the Python `parser` module. The `print_parser_tree` variable controls this mode in the Python2C source code. Below is an example of what an execution of Python2C looks like in this mode. (The output has not been included in its entirety as it is too lengthy to include here. The line “. . .” indicates where the output has been edited. For the complete output you are hereby encouraged to run the Python2C program with the “-ptree” option activated.) The parser tree is more complicated and contains more nodes than the Python abstract syntax tree. This is an example using a very simple “Hello, world!” program. The source code of this program is found in chapter 4.1.3. Even though the code is short and simple, the parser tree does not look short or simple to human eyes.

The Python AST excludes the code comments completely. These are present in the parser tree and can be picked out and used for giving additional commands to Python2C. I have not implemented any support for this in Python2C, as my intention has always been to investigate how to translate the Python language as it is, without making any amendments to it or trying to change the language itself.

This is an execution example showing the command line output of Python2C when the "-ptree" option is active. The line ". . ." indicates where a section of the output has been removed.

```
### Python2C version 0.012
### Programmed by Tom Simonsen (tom.simonsen@gmail.com).
###
### Type in a file name or various options below.
### Type "-help" to display options.
### ->: -ptree
### Printing of the parse tree activated.
### ->: hw.py
### Parsing and translating file "..\Tests\hw\hw.py".

########################################
### The complete parser syntax tree:


[257,
 [266,
  [267,
   [268,
    [280,
     [281,
      [1, 'import'],
      [286,
       [284, [287, [1, 'sys']]],
       [12, ','],
       [284, [287, [1, 'math']]]]]]],
   [4, '# load system and math module']]],
 [266,
  [267,
   [268,
    [269,
     [320,
      [298,
       [299,
        [300,
         [301,
          [303,
           [304,
            [305,
...
                                [311, [1, 's']]]]]]]]]]]]]]]],
                  [8, ')']]]]]]]]]]]]]]]],
   [4, '']]],
 [4, ''],
 [0, '']]
### Debug info files written.
### Additional source files written.
### Code translated and written to files.
###
### About the execution:
```

```
### Nodes in the AST: 35
### Total number of TST nodes created (All may not be part of the TST
structure.): 41
### Parsing of the AST and creation of the TST completed in 0.0789 seconds.
### Translation completed in 0.0007 seconds.
###
### ->:
```

4.1.1.2 The global variables

Many of the variables that are a part of Python2C's main code has already been described in the previous chapter. These are the variables that control the various modes of Python2C, and the value of these variables can be set and altered through the simple textual user interface. There are a few other global variables in Python2C, and this chapter aims to identify and describe their functions.

The version of the Python2C software is set in a variable called `python2c_version`. As exemplified through the command line examples from the previous chapter, the value of this variable is printed each time Python2C starts. It is also amended to every source code file generated by Python2C. This is to inform the user of the version of the software. The user can expect different translation results from the different Python2C versions. The later the version the more functionality has been added to the software, but the user can also expect more bugs and unexpected behavior from the software the more complicated the underlying code is.

The `AST_counter` variable helps to add up the number of nodes in the Python AST during the parsing stage of Python2C. Each of the visitor functions in the `VisitorFunctions` class increases the value of this variable with 1. As the parsing stage is over, the `AST_counter` variable holds the total number of nodes in the Python AST.

The `TST_counter` variable is closely linked to the "-tstnodes" option that is available through Python2C's textual interface. This variable helps to add up the number of TST nodes that are created during Python2C's parsing stage in the same way as the `AST_counter` counts the number of nodes in the Python AST. None of these counters are actually used for anything more than just counting the number of nodes. In this regard they only provide the user of Python2C with statistics. It does not matter a great deal exactly how many nodes there are in the Python AST, for instance. Python2C does not utilize this knowledge in any way during the execution. The nodes are counted and the numbers are printed. That is it. As mentioned earlier, the `TST_counter` does not hold an accurate number of the nodes in the TST, it holds the number of TST nodes that are created during execution. Most of these nodes do end up in the TST structure, but not all. Some TST nodes that are created during execution are intermediate and do never become a part of the TST data structure.

`parse_time` and `trans_time` are two variables that are used to time the execution of Python2C. The results of this timing is output to the command line after each code translation is done. Examples of what this looks like is given in the previous chapter. `parse_time` times the parsing stage of Python2C and `trans_time` times the translation stage. The timing feature of Python2C is not very scientific, by any standard. It measures how long the execution takes in seconds. It completely disregards the fact that many other processes probably are running on the same computer. As a result, the timing results may vary when Python2C translates the same code more than once. The timing feature was included mostly for fun on my part, and it can be upgraded to time the execution more accurately in the future.

The global part of the code also holds a number of file names. This includes the name of the Python file that is going to be translated, the files that contain the resulting translation and the files that contain the various debugging information. The `python_file_name` variable holds the name of the Python file that contains the code that is going to be translated. `cpp_file` holds the file that contains the resulting C++ translation. This may not be the entire resulting C++ code, however. The resulting translation could easily comprise more than one file. `cpp_file` will always hold the C++ file that holds the main function. If the original Python code contains class structures, the resulting C++ translation will have one file for each class in addition to the one file referred to by `cpp_file`. (For more on the translation of classes see chapter 4.3.5.) More than one file will also be created when translating a Python program that has a GUI. These additional files will be held in the variables `gui_header_file` and `gui_resource_file`. (More on the topic of translating GUIs is found in chapter 4.2.3.) The debugging information is written to the files that are held by the variables `v_stack_file`, `var_stack_file`, `func_table_file`, `ast_file` and `exceptions_file`. These files contain debugging information about the visitor stack, variable table, function table, Python AST and exceptions that have occurred during execution, respectively.

Below is a table that shows the files that are output to the destination folder by Python2C. The left column contains the variables in the source code that hold the corresponding file name in the right column. The "" indicates that this part of the file name will be the same as the name of the Python source code file, without the extension. If the Python source code file is named "hw.py", then "hw" will replace the "*".*

Variable	File name
`python_file_name`	*.py
`cpp_file`	*.cxx
`gui_header_file`	resource.h
`gui_resource_file`	*.rc
`v_stack_file`	*_visitor_stack.txt
`var_stack_file`	*_var_stack.txt
`func_table_file`	*_func_table.txt
`ast_file`	*_AST.txt
`exceptions_file`	*_exceptions.txt

4.1.1.3 The global functions

The global part of the Python2C code contains a few functions that set the program in motion. These functions tie the structure of the program together and provide the user with an interface.

`printHelp` simply prints the help menu to the command line. This function is called from the `startCommandLineMenu` function whenever the user enters the "-help" option. See chapter 4.1.1.1 for an example of what kind of output the `printHelp` function produces.

The `parseFile` function is responsible for parsing and translation. It sets up all the data structures that are required, creates names for most of the files that are going to be created and starts the parsing and translation by calling the appropriate functions based on some of the options entered into Python2C by the user.

`checkPaths` checks that the paths and file names that are given to Python2C by the user are valid. This basically means that this functions checks three things. The path to the source folder and the destination folder have to be existing paths. There also has to exist a file name matching the file name the user has given for the file to be translated. This file has to exist in the source folder. If one or more of these checks fail, the `checkPaths` function will return 0. If both paths are valid and the file exists, the function returns 1. This function is called by the `parseFile` function with each new parse or translation.

`startCommandLineMenu` is the first function to be called at the very beginning of the Python2C execution. This function creates and displays the programs textual user interface. It provides the user with the controls to the Python2C software. This function checks all the user input and passes it on to other parts of the program as necessary. It checks that all the commands given by the user are correctly formed and outputs error messages when the user makes mistakes.

4.1.2 The `LanguageConstruct` class and subclasses

The `LanguageConstruct` class is a superclass to all classes of nodes that make up the TST. There is one class of node for each language construct in the Python language. There is also an equivalent relationship to the functions found in the `VisitorFunctions` class. For each function in the `VisitorFunctions` class there exists a corresponding node subclass of the `LanguageConstruct` class. Each of these subclasses contain its own definition of a function named `getString`. This is the function that is producing the C++ code translation for the code that is represented by an instance of one of these classes. When the translation process is started, the `getString` function of the root node in the TST is called first. This function calls the `getString` functions in the child nodes being pointed to by the root node. The child nodes `getString` functions call `getString` functions further down the TST, and so on. The entire TST is, in other words, traversed by calls to `getString` functions in the TST nodes. Each TST node is responsible for producing its part of the C++ translation. All the strings are gathered and output to the C++ source code files.

All the names of the subclasses to the `LanguageConstruct` class bear the same name as in the Python `compiler.ast` module documentation. The only difference is that I have added the string “Lang” to the beginning of each class name, making “Add” “LangAdd”, “And” “LangAnd” and so on. I have chosen to do this to make it clear that these are new classes of objects that contain different and additional information compared to the original Python counterparts. The new objects are a part of a new syntax tree structure that is an altered and decorated version of the original Python AST. New classes of objects have been added and additional information is stored in modified versions of the original Python AST node objects.

There are two special classes of nodes that have been added to make the translation to C++ a little easier. These classes do not have a corresponding visitor function. These two added classes are `LangCreate` and `LangMain`. An instance of `LangMain` is meant to represent the `main` function in the translated C++ code. A Python program may have no `main` function, and as such, this class helps to contain all the code that should be put in the `main` function in the resulting translation.

The `LangCreate` class of objects is designed to hold created variables. These objects are a part of translating variable creation, as variable creation is treated differently in Python and C++. More on the problem of variable creation can be found in chapter 4.3.1.

The table below shows the relationship between all the `LanguageConstruct` subclasses and the visitor functions. It also points out the classes and visitor functions that have been implemented in the current version of Python2C. A "Yes" in the rightmost column does not mean that the class and its corresponding visitor function has been fully implemented. A lot of work may still need to be done. As for the ones with a "No" in the rightmost column, they do exist in the source code. The classes and corresponding functions are defined. Objects of the "No"-classes are never created and the visitor function does not really do anything.

Subclasses of `LanguageConstruct`	**Visitor functions**	**Implemented**
`LangAdd`	`visitAdd`	**Yes**
`LangAnd`	`visitAnd`	No
`LangAssAttr`	`visitAssAttr`	**Yes**
`LangAssList`	`visitAssList`	No
`LangAssName`	`visitAssName`	**Yes**

. . .

. . .

LangAssTuple	visitAssTuple	No
LangAssert	visitAssert	No
LangAssign	visitAssign	**Yes**
LangAugAssign	visitAugAssign	No
LangBackquote	visitBackquote	No
LangBitand	visitBitand	No
LangBitor	visitBitor	No
LangBitxor	visitBitxor	No
LangBreak	visitBreak	No
LangCallFunc	visitCallFunc	**Yes**
LangClass	visitClass	**Yes**
LangCompare	visitCompare	**Yes**
LangConst	visitConst	**Yes**
LangContinue	visitContinue	No
LangCreate		**Yes**
LangDict	visitDict	**Yes**
LangDiscard	visitDiscard	No
LangDiv	visitDiv	**Yes**
LangEllipsis	visitEllipsis	No
LangExec	visitExec	No
LangFor	visitFor	**Yes**
LangFrom	visitFrom	**Yes**
LangFunction	visitFunction	**Yes**
LangGetattr	visitGetattr	**Yes**
LangGlobal	visitGlobal	No
LangIf	visitIf	**Yes**
LangImport	visitImport	**Yes**
LangInvert	visitInvert	No
LangKeyword	visitKeyword	No
LangLambda	visitLambda	No
LangLeftShift	visitLeftShift	No
LangList	visitList	**Yes**
LangListComp	visitListComp	No
LangListCompFor	visitListCompFor	No
LangListCompIf	visitListCompIf	No
LangMain		**Yes**
LangMod	visitMod	**Yes**
LangModule	visitModule	**Yes**
LangMul	visitMul	**Yes**
LangName	visitName	**Yes**
LangNot	visitNot	No
LangOr	visitOr	No
LangPass	visitPass	No
LangPower	visitPower	No
LangPrint	visitPrint	No
LangPrintnl	visitPrintnl	**Yes**
LangRaise	visitRaise	No
LangReturn	visitReturn	**Yes**
LangRightShift	visitRightShift	No

LangSlice	visitSlice	**Yes**
LangSliceobj	visitSliceobj	No
LangStmt	visitStmt	**Yes**
LangSub	visitSub	**Yes**
LangSubscript	visitSubscript	**Yes**
LangTryExcept	visitTryExcept	No
LangTryFinally	visitTryFinally	No
LangTuple	visitTuple	**Yes**
LangUnaryAdd	visitUnaryAdd	No
LangUnarySub	visitUnarySub	No
LangWhile	visitWhile	**Yes**
LangYield	visitYield	No

4.1.3 The `VisitorFunctions` class

The `VisitorFunctions` class holds all the visitor functions that are called by the `walk` function in the `compiler.visitor` module. `walk` is responsible for walking the Python AST structure. In Python2C this is done by calling the `walk` function and supplying the root node of the Python AST as an argument. `walk` will then call the appropriate visitor function for each of the nodes pointed to by the root node. The name of the function it tries to call is “visit” followed by the name of the Python class that the node belongs to. If the node is an object of the class `Assign`, for instance, the `visitAssign` function will be called, if it exists. The visitor functions will in their turn call the `walk` function and supply the root of the current subtree as argument. This process goes on until the entire syntax tree has been traversed and all the nodes have been visited. The walk of the Python AST is output to the “`*_AST.txt`” file. Below is an example of such a file and the Python source code is was derived from. The numbers on the left in the file indicate the number of nodes in the AST. For this particular source code the AST is made up of 35 nodes.

This is a simple Python “Hello, world!” program contained in a file called “hw.py”.

```
#!/usr/bin/env python
import sys, math       # load system and math module
r = t = float(sys.argv[1]) # extract the 1st command-line arg.
s = 1 + math.sin(r)
u = ""
print "Hello, World! sin(" + str(r) + ")=" + str(s)
```

This is the resulting contents of the “hw_ast.txt” file.

```
########################################
### The complete compiler AST:

Module(None, Stmt([Import([('sys', None), ('math', None)]), Assign([AssName('r',
'OP_ASSIGN'), AssName('t', 'OP_ASSIGN')], CallFunc(Name('float'),
[Subscript(Getattr(Name('sys'), 'argv'), 'OP_APPLY', [Const(1)])], None, None)),
Assign([AssName('s', 'OP_ASSIGN')], Add((Const(1),
CallFunc(Getattr(Name('math'), 'sin'), [Name('r')], None, None)))),
Assign([AssName('u', 'OP_ASSIGN')], Const('')),
Printnl([Add((Add((Add((Const('Hello, World! sin('), CallFunc(Name('str'),
[Name('r')], None, None))), Const(')='))), CallFunc(Name('str'), [Name('s')],
None, None)))], None)]))

########################################
### The walk of the AST:

1 visitModule
      |--> (None, Stmt([Import([('sys', None), ('math', None)]),
Assign([AssName('r', 'OP_ASSIGN'), AssName('t', 'OP_ASSIGN')],
CallFunc(Name('float'), [Subscript(Getattr(Name('sys'), 'argv'), 'OP_APPLY',
[Const(1)])], None, None)), Assign([AssName('s', 'OP_ASSIGN')], Add((Const(1),
CallFunc(Getattr(Name('math'), 'sin'), [Name('r')], None, None)))),
Assign([AssName('u', 'OP_ASSIGN')], Const('')),
Printnl([Add((Add((Add((Const('Hello, World! sin('), CallFunc(Name('str'),
[Name('r')], None, None))), Const(')='))), CallFunc(Name('str'), [Name('s')],
None, None)))], None)]))
2 visitStmt
      |--> (Import([('sys', None), ('math', None)]), Assign([AssName('r',
'OP_ASSIGN'), AssName('t', 'OP_ASSIGN')], CallFunc(Name('float'),
[Subscript(Getattr(Name('sys'), 'argv'), 'OP_APPLY', [Const(1)])], None, None)),
Assign([AssName('s', 'OP_ASSIGN')], Add((Const(1),
CallFunc(Getattr(Name('math'), 'sin'), [Name('r')], None, None)))),
Assign([AssName('u', 'OP_ASSIGN')], Const('')),
Printnl([Add((Add((Add((Const('Hello, World! sin('), CallFunc(Name('str'),
[Name('r')], None, None))), Const(')='))), CallFunc(Name('str'), [Name('s')],
None, None)))], None))
3 visitImport
      |--> ([('sys', None), ('math', None)],)
4 visitAssign
      |--> (AssName('r', 'OP_ASSIGN'), AssName('t', 'OP_ASSIGN'),
CallFunc(Name('float'), [Subscript(Getattr(Name('sys'), 'argv'), 'OP_APPLY',
[Const(1)])], None, None))
5 visitAssName
      |--> ('r', 'OP_ASSIGN')
6 visitAssName
      |--> ('t', 'OP_ASSIGN')
7 visitCallFunc
      |--> (Name('float'), Subscript(Getattr(Name('sys'), 'argv'), 'OP_APPLY',
[Const(1)]), None, None)
8 visitName
      |--> ('float',)
9 visitSubscript
      |--> (Getattr(Name('sys'), 'argv'), 'OP_APPLY', Const(1))
10 visitGetattr
      |--> (Name('sys'), 'argv')
11 visitName
      |--> ('sys',)
12 visitConst
      |--> (1,)
13 visitAssign
      |--> (AssName('s', 'OP_ASSIGN'), Add((Const(1),
CallFunc(Getattr(Name('math'), 'sin'), [Name('r')], None, None))))
14 visitAssName
      |--> ('s', 'OP_ASSIGN')
15 visitAdd
      |--> (Const(1), CallFunc(Getattr(Name('math'), 'sin'), [Name('r')], None,
None))
16 visitConst
      |--> (1,)
17 visitCallFunc
      |--> (Getattr(Name('math'), 'sin'), Name('r'), None, None)
18 visitGetattr
      |--> (Name('math'), 'sin')
```

```
19 visitName
      |--> ('math',)
20 visitName
      |--> ('r',)
21 visitAssign
      |--> (AssName('u', 'OP_ASSIGN'), Const(''))
22 visitAssName
      |--> ('u', 'OP_ASSIGN')
23 visitConst
      |--> ('',)
24 visitPrintnl
      |--> (Add((Add((Add((Const('Hello, World! sin('), CallFunc(Name('str'),
[Name('r')], None, None))), Const(')='))), CallFunc(Name('str'), [Name('s')],
None, None))), None)
25 visitAdd
      |--> (Add((Add((Const('Hello, World! sin('), CallFunc(Name('str'),
[Name('r')], None, None))), Const(')='))), CallFunc(Name('str'), [Name('s')],
None, None))
26 visitAdd
      |--> (Add((Const('Hello, World! sin('), CallFunc(Name('str'), [Name('r')],
None, None))), Const(')='))
27 visitAdd
      |--> (Const('Hello, World! sin('), CallFunc(Name('str'), [Name('r')],
None, None))
28 visitConst
      |--> ('Hello, World! sin(',)
29 visitCallFunc
      |--> (Name('str'), Name('r'), None, None)
30 visitName
      |--> ('str',)
31 visitName
      |--> ('r',)
32 visitConst
      |--> (')=',)
33 visitCallFunc
      |--> (Name('str'), Name('s'), None, None)
34 visitName
      |--> ('str',)
35 visitName
      |--> ('s',)
```

For every visitor function there is a matching class to be found in the subclasses of the `LanguageConstruct` class, but there is no one-to-one relationship between the functions and the classes. There are only so many visitor functions as there are types of objects in the Python AST. All the added classes of objects that are a part of the TST (and not part of the Python AST) have no corresponding visitor function. The creation of the objects of the additional TST classes are taken care of within the visitor functions. `visitStmt`, for instance, is responsible for creating `LangStmt` objects and adding them to the TST structure. In some cases it has to create a `LangMain` object instead.

The `VisitorFunctions` class does not only contain the visitor functions. It also holds a few other important elements of the Python2C program. The first and perhaps the most important one is the pointer to the `VStack` instance, stored in the variable `v_stack`. The `VStack` is described in detail in chapter 4.1.5.

The second element is a pointer called `global_constructs`. This is a pointer that always points to the TST node that represents the main function, an instance of the `LangMain` class. Easy access to the main node is quite useful since new additions to the main function's contents are made quite frequently during the traversal of the AST. A direct pointer is more efficient than having to do a search of the TST each time the `LangMain` object is needed. The "main code" in a Python program can be found in many locations in the source code, since this part of the code does not have to be contained within something like a class or a function. C++, on the other hand, requires the main code to be put in a function called "main". The `LangMain` object also holds information about the overall structure of the translated program. All include statements, classes, global variables and forward declarations of functions are pointed to by this object.

Third, the `VisitorFunctions` class holds a number of flags that are used to check what part of the code an AST `Stmt` object is part of. The `Stmt` object contains all statements in a code block, the part of the code that you would find within brackets "`{}`" in C++. The brackets are replaced by indentation of the lines of the code statements in Python. To be able to easily determine if the statements the `Stmt` object points to are part of a class, a function or a conditional structure (else, if, for and while) flags are set and checked by `visitClass`, `visitFunction`, `visitWhile` and other functions. These flags make it easier to determine what part of the code an `Stmt` object is part of, and appropriate action can be taken on the basis of this information. These flags are `inside_class`, `inside_function` and `inside_conditional`.

The flags are also useful for debugging. The variable and function tables can be validated more easily when additional information in the form of the flags are present. It is easier to check if a variable or function is global or part of a class, for instance, and if it has been added to the tables correctly by Python2C.

4.1.4 The `Stack` class

This is a simple implementation of a stack data structure by using the Python language's built-in list structure. It contains the common stack functions `push` and `pop` that add an object to the top of the stack and remove an object from the top of the stack, respectively. In addition, this class also contains the function `getTop` that returns the object at the top of the stack. The function `isEmpy` returns 1 if the stack does not contain any objects and 0 if it contains objects. Instances of the `Stack` class are used to make up the `VariableTable` class' data structure.

4.1.5 The `VStack` class

General

This class is very similar to the `Stack` class and the name "VStack" is short for "Visitor Stack". Only one instance of this class is created during the execution of Python2C and the `v_stack` variable in the `VisitorFunctions` class points to this object. This `VStack` object is a vital part of the data structure that is used during the traversal of the Python AST, and it serves as the parsing stack for the creation of the TST. Every syntactic element or AST node is pushed onto the top of it as the AST is being traversed. When the parsing of the particular syntactic element is done, the element is popped off the visitor stack.

The top element of this stack is used to build the "translated syntax tree", or TST, structure. When parsing a sub-element of a syntactic structure the parent structure is always the top element of this stack. The child element is then linked to its parent in the TST structure before the child element is pushed on top of the stack. The resulting TST structure is very similar to the Python AST structure, that is obtainable through the `compiler.ast` module. The TST is created from traversing the Python AST, and the AST and the TST are therefore closely related. The TST is an altered and decorated version of the Python AST. The translated C++ source code is translated from the TST structure, and all the extra information is added to the TST to aid this translation.

The internal functions

The `VStack`'s functions are basically identical to the ones of the `Stack` class. The only difference is that the `VStack`'s `pop` function writes debugging information to the "`*_visitor_stack.txt`" file in addition to popping the top element. Each time the `VStack` is popped, the textual representation of the contents of the entire stack is added to this file.

4.1.6 The `VariableTable` class

General

An instance of the `VariableTable` class makes up Python2C's internal symbol table structure together with an instance of the `FunctionTable` class. The `VariableTable` object maintains a stack of all variables reachable from the scope that is currently being parsed by the code. When the parsing stage is finished the `VariableTable`'s stack contains no variables.

The data structure

The `VariableTable` class is made up of two separate internal stack structures. The first and most important of these two is the one made up of a list of dictionaries, using the built-in Python list and dictionary structures. The top of this stack contains a dictionary containing all the variables declared in the current scope of the code. The names of the variables are the keys in this dictionary and the `LanguageConstruct` objects are the values. The variables created outside the current scope are stored in the stack elements below the top.

The second internal stack is an instance of the `Stack` class. Each stack element contains a list of the values of the `VisitorFunctions`' `inside_class`, `inside_function` and `inside_cond` flags. The top element of this stack corresponds to the top element of the first stack. This part of the data structure keeps track of where the variables where created. For instance, if the variables were created inside a function inside a class, both the `inside_class` and the `inside_function` flags would be set to 1 and `inside_cond` set to 0.

This class implements the "most closely nested" rule. This rule states that variables that are not created in the same scope can use the same names. The scope in which a variable is created, together with its name, identifies the variable uniquely. When the `VariableTable` is asked to look up a variable by name it checks the current scope for a variable with this name. If it exists, then this is the variable that is looked up. If no variable with this name exists in the current scope the search continues down the stack until a variable with a matching name is found or until the bottom of the stack is reached.

The internal functions

In addition to the usual `push` and `pop` functions this class contains the functions `find`, `getType`, `getValue`, `getObject`, `addToTop`, `getTop`, `isEmpty` and `getFlagList`. The `find`, `getType`, `getValue` and `getObject` functions searches the stack for a variable with a matching name according to the most closely nested rule. If found the "find" function returns the value 1, if it is not found the value 0 is returned. `getType` returns the type of the variable if it exists. If the variable is not found it returns the empty string. `getValue` and `getObject` both return objects or none if the variable does not exist. `getValue` returns the value currently stored in the variable. This is not fully implemented in Python2C and this function is not in use. `getObject` returns the variable object stored in the `VariableTable` stack.

`addToTop` is not the same as the push function. The push function pushes an entirely new dictionary to the top of the stack while `addToTop` adds a new entry to the already existing dictionary at the top of the stack.

`getTop` returns the top element of the `VariableTable`. `isEmpty` returns the value 1 or 0 based on whether the stack is empty or not, respectively. `getFlagList` returns the entire internal stack containing the flags.

4.1.7 The `FunctionTable` class

General

The `FunctionTable` class differs a bit from the `VariableTable` class. The `VariableTable` only provides temporary storage for the variables during parsing of the code. After the parsing process is over the `VariableTable` is empty and no longer in use. The `FunctionTable`, however, is set up during the parsing of the Python source code, and its contents continue to exist after the parsing has ended. Its job is to maintain a table of all classes and functions in the parsed program. It also maps the functions to the classes that they belong to. A function that is not declared inside a class belongs to the "Global level". All the static functions, the ones that are declared outside classes, are translated to be a part of this level.

The data structure

The `FunctionTable` also differs from the `VariableTable` when it comes to the structuring of the data. Where the `VariableTable` is implemented as a stack structure the `FunctionTable` actually has more of a table structure. It consists of two internal dictionaries called `function_table` and `class_table`. The first one of these is a dictionary containing dictionaries, with the class names as keys. Each dictionary entry in the `function_table` contains a dictionary that contains all the functions that have been declared in the source code of the class in question. The names of the functions are the keys in this dictionary and the values are pointers to the corresponding `LangFunction` objects. The `class_table` is a dictionary that contains a mapping between the class names and their corresponding `LangClass` objects.

The internal functions

The `FunctionTable` contains the functions `addClass`, `getClass`, `getClassFunctions`, `addFunction` and `getFunction`. Since the `FunctionTable` is not implemented as a stack, it does not contain the usual `push` and `pop` functions.

The names for the functions are relatively self-explaining. The `addClass` function adds a TST class object (an instance of the `LangClass` class) to the `FuctionTable` structure. In the internal function_table, a new entry is added with the name of the class as the key. An empty dictionary that will hold the functions of the class is set as the value of this new entry. These functions can be added using addFunction explained in more detail below. addClass also adds a new entry to the internal class_table. The key of this entry is the class name and the value is the TST LangClass object.

The getClass function is used to look up the name of a class in the FunctionTable. If it finds a matching name among the keys of the class_table, the object stored in the corresponding value in this dictionary will be returned.

getClassFunctions is also a look-up function, used to return all the functions of a class. If this function finds a matching class name in the function_table, the entire dictionary stored in this class name's entry is returned.

addFunction adds a function object to the FunctionTable by adding it to the internal function_table dictionary. To be able to add a function's object to the FunctionTable's structure you have to provide addFunction with the name of the class that the function belongs to. If the function does not belong to any class and is a global or static function, it has to be added to the "Global level". The string "Global level" is used as the "class name" in this case. The "Global level" entry in the FunctionTable is always present in the structure, and adding a function to it is not any different from adding a function to a class.

Known weaknesses

There is currently no implemented support for subclasses in Python2C. The structure of the FunctionTable does support this, however. All LangClass objects contain information about inheritance in the form of the names of the parent classes. The parent classes and their functions can therefore be looked up in the FunctionTable.

4.1.8 The `Translator` class

The Translator class is responsible for overseeing the translation process. It holds what I have chosen to name the *TST* (Translated Syntax Tree) and various translation tables and dictionaries. In Python2C there is no TST class, a value in the TST list in the Translator class points to the top element in the TST node structure that is created in the parsing stage of the Python2C execution. The Translator class contains functions for searching the TST in various ways and functions for correctly adding variables to the variable table.

4.1.8.1 The translated syntax tree (TST)

The reference to the top TST node is held in the first element of a list named TST. The reason that this variable is a Python list structure and not a regular pointer is that an implementation using a list opens the possibility for storing more than one TST during the Python2C execution. Using more than one syntax tree structure during parsing or translation can be useful for many reasons. It can be used to keep back-ups of different versions of the same TST as the TST is altered as the program is executing. Many compilers use different stages in the compilation process. In some implementations the reason for keeping these separate is that it is better if the stages do not overwrite the same data structures or files. Many compilers also utilize the different stages to create different versions of the syntax tree. The first parsing stage of the compiler may create the first and preliminary version of the syntax tree, a second parsing stage may create a “decorated” version of the syntax tree that contains extra information of different sorts, a third stage may traverse the decorated version of the syntax tree to produce a more optimal version of it and so on. This optimization is usually performed to make the resulting target code more efficient.

4.1.8.2 The abstract syntax tree (AST)

The current implementation of Python2C does not take advantage of a multiple stage strategy to manipulate the TST, but code that uses this feature can be added at some later point, without having to make changes to the existing first-stage code. The translation process executed by Python2C can be seen as having more than one stage, never the less. The first stage of the process is executed by the Python compiler. It scans and parses the source code to create an abstract syntax tree (AST), which can be obtained through Python's `compiler` modules.

In Python2C the traversal of the AST to create the TST is done by using the `walk` function found in the `compiler.visitor` module, as already mentioned. This function is responsible for traversing the AST correctly from top to bottom. All the visitor functions reside in the `VisitorFunctions` class, which is documented in chapter 4.1.3.

4.1.8.3 The variable table

The `Translator` class holds the pointer to the variable table, which is named `stack_of_variables` because of its internal data structure (See chapter 4.1.6 for a detailed description of this structure). Every time the Python2C encounters a variable in the code it is translating, it consults the variable table to check if the variable exists or not. The consulting itself is done through calling two functions in the `Translator` class. The first one of these is `checkVariable`. This function consults the variable table directly and takes the appropriate action depending on whether the variable exists in this table or not. If the variable does not exist, the `createVariable` function is called to create the variable and add it to Python2C's data structures correctly.

Python2C generates a few extra variables in some cases. These are variables that do not exist in the original Python source code but have to exist in the translated C++ source code to make the translated program work. These variables are typically created whenever a for loop is being translated. (This is explained further in chapter 4.3.3.) To avoid creating a new variable with a name that is already in use and to avoid conflicts the `checkVariable` function also ensures that a created variable receives a unique name that is not in use by the Python program that is being translated. Firstly, the name that is suggested for the new variable is fairly long and is therefore less likely to be in use. When you have two or more while or for loops occurring in the same scope, variable names are most likely to already be in use. This is because Python2C will suggest the same name for the variable each time. In this case a number is added at the end of the name starting on 1 and increasing with the number of conflicting names.

When a variable name is completely new, meaning that it does not exist in the variable table, `checkVariable` calls `createVariable`. `createVariable`, which the name suggests, is responsible for adding the variable object to the variable table. It is also responsible for determining the type of the variable, if this is possible at this stage in the code.

In some cases the type of a variable can be very hard to determine. The more complex an assignment statement is, the harder it is to determine the type. Another case, that occurs frequently, is when the part of the statement that assigns the value refers to a part of the code that has not yet been parsed by Python2C. In this case the `createVariable` function has no way of determining the type and the determination has to be done at some later point in the translation process. The current version of Python2C has a very limited support for checking the type later. The best way to make sure that the types of all variables have been determined correctly is to do a second traversal of the syntax tree. In practice, this would be done by a second pass of the TST. As mentioned earlier in this thesis, Python2C supports multiple passes of the syntax tree structure but it remains to be implemented.

4.1.8.4 The function table

Python2C's function table structure is pointed to by `function_table` in the `Translator` class. Unlike the checking and creation of variables described above, the `Translator` class has no functions for controlling the checking and creation of functions. All the checking is done by code found in some of the `LanguageConstruct` classes, some of the `VisitorFunctions` functions and the code in the `FunctionTable` class. The data structure of the function table is made up of an instance of the `FunctionTable` class, described in more detail in chapter 4.1.7.

4.1.8.5 The various translation tables

The `Translator` class contains quite a few translation tables that are implemented as Python dictionaries. These tables can roughly be divided into two categories: The ones that translate something and the ones that keep track of something. The ones that translate something are usually filled out by hand, and the ones that keep track of something are usually filled out by Python2C as it is running. The important variable table and function table are also pointed to by variables in the `Translator` class.

The `module_table` contains the modules that can be translated or partially translated by an `#include` in the translated C++ code. Python's `math` module is a good example. This module is very similar to `math.h`, which is the C++ equivalent. Both the `math` module and `math.h` have the same purpose and functionality. The functions correspond and have the same names to a good extent. If the `math` module is imported in the Python code, then `math.h` should be included in the translated C++ code. The translation of a complete Python module is of course not as easy as just importing the C++ equivalent. Each function in the module may have to be translated as well. In some cases it is enough to have a table that lists what each of these functions return. This is largely the case with the `math` module. In other cases the translation of a module function has to be done by writing the function by hand in C++ code and including this C++ code as a part of the translated target code.

imported_modules and imported_modules_from are two tables that keep track of the modules that have been imported in the Python code. imported_modules keeps track of the modules that have been imported using the import statement. imported_modules_from keeps track of the ones that have been imported using the from statement. The reason to separate these two import tables is the fact that the two import statements have different meanings. When a standard import statement is used to import a Python module, the functions in the module can only be referenced by referring to the imported module. The pow function in the math module, for instance, can only be referenced by math.pow, if this is the case. If the math module is imported through a from statement, its pow function can now be referenced directly, without using the name of the module. A call to pow will call the math module's pow function. The two tables make it easier to decide what function is being called in the Python code and to keep track of the modules that were imported in the one or the other way.

standard_includes is a list of all the C++ #include directives that are always present in the translated code, regardless of the program that is being translated. A list of all the #includes that are generated by Python2C's traversal of the AST is held in generated_includes. This is a list of all the #includes that must be a part of the C++ code in addition to the standard includes, to make the translation complete.

The types dictionary is also among the translation tables, containing a complete list of all supported Python2C types that are used in the TST. Most of the types in this dictionary correspond to types found in Python, but they have their own Python2C name to distinguish them from their Python equivalents. These names are all written in capital letters. The Python2C types are listed as the keys in the dictionary and the values are the corresponding types found in C++. Not all of the Python2C types in the types table have C++ or real Python equivalents. This is true for all the types related to the translation of the Tkinter module. More on this is found in the chapter dealing with GUI translation (chapter 4.2).

Below is the dictionary types as it appears in the Translator class.

```
# This dictionary contains a mapping between the string representations used for
# types in this program and the string representations used in C++.
types = {"FLOAT":       "double",
         "INT":         "int",
         "STRING":      "char*",
         "VOID":        "void",
         "LIST":        "struct List*",
         "TUPLE":       "struct Tuple*",
         "DICT":        "struct Dict*",
         "LISTNODE":    "struct ListNode*",
         "TUPLENODE":   "struct TupleNode*",
         "TOPLEVEL":    "TOPLEVEL",
         "FRAME":       "FRAME",
         "BUTTON":      "BUTTON",
         "CANVAS":      "CANVAS",
         "CHECKBUTTON": "CHECKBUTTON",
         "ENTRY":       "ENTRY",
         "LABEL":       "LABEL",
         "LISTBOX":     "LISTBOX",
         "MENU":        "MENU",
         "MENUBUTTON":  "MENUBUTTON",
         "RADIOBUTTON": "RADIOBUTTON",
         "SCALE":        "SCALE",
         "SCROLLBAR":    "SCROLLBAR",
         "TEXT":         "TEXT",
         "NOTYPE":       "/* Python2C Error NOTYPE */",
                        # The type should never be of type "NOTYPE"
                        # at the point of translation.
         "":             "/* Python2C Error \"\" */"
                        # The type should never be the empty string
                        # at the point of translation.
        }
```

`tkinter_types` is a dictionary that contains all of the types related to the translation of the `Tkinter` module. This is basically just a look-up table, used to check if a type is a `Tkinter` type. All the `Tkinter` types are also listed in the `types` dictionary, as you can see above. `Tkinter` types do not really exist in the Python language and is something uniquely created for Python2C to make it easier to translate GUIs. When an object can be identified as a part of a program's GUI, it makes it easier to keep the GUI translation less dependent on the rest of the code. All objects that have some function in the GUI are labeled and their function in the GUI is pointed out by their type.

Below is the dictionary `tkinter_types` as it appears in the `Translator` class. The values for all the Tkinter types are empty strings as there exists no C++ equivalents.

```
tkinter_types = {"TOPLEVEL":    "",
                 "FRAME":       "",
                 "BUTTON":      "",
                 "CANVAS":      "",
                 "CHECKBUTTON": "",
                 "ENTRY":       "",
                 "LABEL":       "",
                 "LISTBOX":     "",
                 "MENU":        "",
                 "MENUBUTTON":  "",
                 "RADIOBUTTON": "",
                 "SCALE":       "",
                 "SCROLLBAR":   "",
                 "TEXT":        ""}
```

The `Translator` class also contain a number of translation tables that list the functions or variables in various Python modules. The keys in these tables are the names of the variables or functions and the values are their types or return types respectively. Some of these tables are `functions_module_math`, `functions_module_tkinter` and `vars_module_sys`. I have created some sort of a naming convention for these tables. As exemplified by the three tables that I have mentioned above, this naming convention is as follows: The first part indicates whether the module contains a mapping of functions and their return types or variables and their types. The second part indicates that it is a module. If this part does not say “`module`” it might say “`python_lang`” instead, indicating that the table contains a mapping for some functions or variables found in the Python language. The third, and last part, names the module in question, pointing out the module that contains the functions or variables. If the second part reads “`python_lang`” then there is no third part.

4.1.8.6 Translation of Python data types

The list, the tuple and the dictionary are data types that are a part of the Python language, but equivalent data types do not exist in the C++ language. If Python2C is going to be able to translate programs that use these data types, these data types must be given their C++ equivalents. I chose to do this by implementing a list, a tuple and a dictionary in pure C, not C++. Coding it in C is a bit more cumbersome, but hopefully it will make the translated code run more efficiently. All of the names of the files containing this C source code are in the `standard_includes` table, described in the previous chapter. The lists, tuples and dictionaries are frequently in use in a typical Python program, so I chose to always have them as a part of the translated C++ code. All the files in the `standard_includes` table are always included in any program translated by Python2C. This can be seen in the many code examples throughout this thesis.

The `source_code_files` list holds the file names that contain the C source code for the above mentioned data structures. These files are stored in the same folder as the Python2C source code. When a program is translated by Python2C, all of these files are copied to the destination folder along with all the other files related to the translation.

The `python2ctypesheader.h` file

The `python2ctypesheader.h` file is included by all the files that comprise the C translation of the Python data types. This file defines the types `NoType`, `IntType`, `DoubleType`, `StringType`, `ListType`, `TupleType` and `DictType` by setting them to values in a C enumeration. The `NoType` value is a value indicating that no type has been set in the C `struct` in question. The C `struct`s `List`, `Tuple` and `Dict` are also defined in this file in addition to some definitions that set the size of the amount of memory that is allocated by certain elements.

`CHAR_PER_NODE1`, `CHAR_PER_NODE2` and `DICT_CHAR_PER_INDEX` are precompiler definitions that can have their values altered according to what is required by the translation task at hand. These three values are a part of determining the size of the string buffers used when printing the textual representation of the contents of the data types. `CHAR_PER_NODE1` is a part of setting the string buffer size for each individual tuple or list node, and `CHAR_PER_NODE2` is a part of setting the size of the string buffer used for each entire `List` or `Tuple`. `DICT_CHAR_PER_INDEX` sets the size of the string buffer used for printing an entire dictionary. The value corresponds to number of characters per index in the dictionary. If there are 1000 indexes in the dictionary and the value of DICT_CHAR_PER_INDEX is 4, the buffer will be 4000 characters long. The `DICT_SIZE` value sets the number of indexes in a dictionary.

The `List`

All of the translated Python types have been coded in C by creating data structures from C `struct`s and pointers. The list structure, contained in the file `python2clist.h` is built up of a singly linked list, where the nodes contain only one pointer to to next node in the list sequence. The list structure has a head pointer that points to the first node in the list and a tail that points to the last node in the list. It also stores its length (its number of nodes) and the type of variables its nodes hold. All the nodes in one list can only store values of the same type. An attempt to add an integer to a list containing doubles would result in an error being generated. The list cannot change its type during execution, although this is fully possible considering the data structure alone.

The list nodes, defined in `python2clistnode.h`, contain storage space for each type of value it can hold. These values are: `int`, `double`, `char*`, `List*`, `Tuple*` and `Dict*`. In addition to the pointers and the values, the nodes hold an integer variable `type` that stores a number defining the type of value stored in the `List` node. This is a value between 0 and 6 that corresponds to the values defined in the enumeration in the file `python2ctypesheader.h`. Lastly, the nodes contain a `char* key` that is only in use when a `List` node is a part of a dictionary structure, explained later. The `python2clistnode.h` file is included by both `python2clist.h` and `python2cdict.h`, since these two structures use the same node `struct`s.

The `Tuple`

The tuple structure has a lot in common with the list structure described in the above section. It is also built using a singly linked list structure where the nodes only contain one pointer to the following node. It has a head and a tail node and keeps track of the number of nodes it holds. The difference is that it does not care about what type of data the nodes hold. A tuple can hold a combination of all types of data in any sequence.

The tuple nodes are defined by the code in the file `python2ctuplenode.h`. This file is included by the `python2ctuple.h` file that defines the functionality of the tuple structure. These nodes are very similar to their list counterparts and the only thing that tells them apart is the fact that the tuple nodes do not have the `key` variable.

You can not append or delete items in a Python tuple. This is indeed possible in my translation of this structure, although a translation from Python to C++ that is using this feature is not possible since it does not exist in Python. In my translation it is also possible to assign a value to a tuple element (such as the statement `tuple[2] = 1`). This is also not possible in the Python language. All in all, my translation of the tuple data type is not as rigid as the Python version of the same structure. That aside, my translation should retain all the functionality of its Python counterpart.

The `Dict`

The C translation of the dictionary data type is implemented as a hash table. The size of this table is set by the `DICT_SIZE` in the `python2ctypesheader.h` file and cannot be changed during execution. The table consists of a single array that contain pointers to `Lists`. The hash function uses the `key` string to compute the index at which the new element is going to be stored. When a new element is added a new `List` is created, and the pointer in the array at the computed index will point to this `List`. The new element is then inserted into this `List` as a `ListNode`. If a `List` is already being pointed to at the computed index, a collision has occurred. Two or more elements are now being stored at the same index in the dictionary. The new element is simply added to the end of this `List` and the number of collisions for the dictionary is increased by one. The `key` variable in the `ListNode` is used to store the key string when the `ListNode` is a part of a `Dict`.

As already indicated, the `Dict stuct` contains a `collisions` variable that stores the number of collisions that has occurred in this dictionary. The fewer indexes a dictionary has, the more likely these collisions become. The standard value of `DICT_SIZE` is 1000, which is quite a small number when considering the norm for the size of the typical hash table structure. This number has mostly been set this low for debugging purposes, so that collisions do occur rather frequently.

The `Dict struct` also stores its own size. In this case the size relates to the number of indexes in the hashed array. The actual number of elements is stored in the `num_elements` variable. The dictionary also has a `type` variable, meaning that it can only store one type of data. The type of the dictionary can not change during execution, making the `Dict` a stricter data structure than the one found in Python.

Currently the `Dict` structure has no `getKeys` function, making it impossible to translate calls to the `keys` function of a Python dictionary. This function can be added rather easily. Python2C will not translate statements like `i = dict["abc"]`, since this feature has not been developed as yet.

4.2 Translating GUIs

I have done a bit of investigating when it comes to translating the Python `Tkinter` module to Windows CE GUI. No implementation of this has been made a part of my Python2C translator, although some of the data structuring that is required is present in the source code in the current version.

My solution is based on the idea that the GUI translation should be done with only one pass of the TST, as is the case with the rest of the translation of Python to C++. As mentioned earlier, Python2C does not currently support multiple passes. The GUI translation should not be any different in this regard, the way I see it. This does require, however, that the GUI code is structured in a certain way for a one pass algorithm to be able to translate as much as possible and make the translated code work. The programmer cannot structure the Python GUI code in any way he wants to or is accustomed to. Why this is and how the code should be structured is discussed in this chapter. This chapter serves as a description of how GUI translation could be automated using Python2C.

4.2.1 The GUI class

When you have only one pass in which to translate the Python GUI there is one fundamental issue that requires attention: *Each GUI element has to be identified before it is referenced*. Each GUI element in the Python source code can be referenced as any other type of variable. If a reference to a variable is placed at line 13 and the creation of this variable is done on line 509, then Python2C will not realize that it is dealing with a reference to a GUI element (or widget) while it is examining the code on line 13. This could lead to an erroneous translation. The Windows GUI code uses a handler function for each displayed dialog. `Tkinter` does not require this. In `Tkinter` a function is normally called based on the action that just took place. It is the GUI element that registered the action that is responsible for taking action. This could amount to several functions being called depending on the actions that a particular widget listens for. In Windows the handler function is always called. It is this function that decides what to do based on the action that just took place. Thus, it is very important for Python2C to know when it is dealing with a GUI element. For the handler function to be called, it is required that the handler function exists in the target code. Python2C has to parse the creation of the GUI element before any reference to it, to be able to generate a handler function definition for that specific GUI element.

Windows GUI programming is what I would label as “dialog-oriented”. The GUI is made up of dialog elements that are typically displayed in separate windows. A dialog usually contains more than one graphical element that each is given its own unique identifier within the program. The dialogs themselves also have unique identifiers. Each dialog has a message handler function that responds to events that occur as the program is running. To be able to translate handler functions well, Python2C has to identify what graphical components belong to what dialog. This can be done by generating a message handler function for each window except pop-ups, as pop-up windows are treated separately by Windows and do not need any. The elements that belong to a window will have to be included in the translated message handler function code. Each such element can be identified as they are added to the layout or frame inside a window. The menu bar of a program does not need its own message handler function. All of the message handling for this element is done in the `WndProc` callback function (`WndClass.lpfnWndProc`). The contents of this function has to be generated by Python2C, however.

It is important for the translation that all the GUI elements are created at more or less the same time. All dialogs and their components have to be parsed and identified before they are referenced. The earlier in the source code this is done the better it is. The ideal thing would be to put all the creation of GUI elements inside one, single class that is as close to the first line of code as possible. This class must be named “GUI” to set it apart from all the other classes that might be found in the program. Python2C would then be able to map out the entire GUI structure and generate all the handler functions and the complete resource script, before any other part of the AST is traversed. This would ensure that all the handler functions and all handles, resources, layouts and unique identifiers exist at the time they are referenced.

All GUI elements would have to be created inside this class. Even the windows and other GUI elements that are not to be used or displayed at this point in the execution of the program, will have to be created here. The root window in a Python program is created by a call to the `Tk` function. All additional windows are created by a call to the `Toplevel` function. These windows will become visible by default. If the purpose of the code is to create the window for future use and not display it right away, a call to this window's `withdraw` function is necessary. A call to the `deiconify` function will display the window again.

As explained in chapter 2.6.1.3, all the layout of the Windows GUI is taken care of in a separate language, known as *resource script*. The resource script or RC script is located in a separate file with a “.rc” file name extension. The RC script defines the layout of each graphical element inside each dialog and uses the unique identifiers to refer to the graphical elements. It takes care of text inside the windows and the formatting of it, although text can also be put in the GUI by the C++ code. The RC script controls the positioning by setting the position for each component in pixels.

4.2.2 Generating the contents of the handler functions

This is where it gets a little tricky. The insides of the translated handler functions would have to be a translated version of the contents of the functions that are called when actions occur, in the Python source code. For this to even be possible, the functions that are called by a GUI element have to be identified.

The easiest way to get around this problem is to apply some naming and coding conventions. Firstly, all the functions that are called by a GUI element have to be named “handler_*”. This is so that Python2C can tell the handler functions apart from other functions in the source code. In addition, all such functions have to have the name of the GUI element following “handler_” in their names. This would be the name of the variable where the GUI element is stored. If a handler function is called by two different GUI elements, then there needs to be two separate definitions of this function, under different names (The variable names following “handler_” would be the only difference between the two functions.), in the Python code. It is important to identify exactly the one

GUI element that uses the handler function through the naming convention. This is one way of matching up the GUI elements and functions that belong together without using multiple passes to track the use of the functions. Two GUI elements that have to use the same function might reside in two different dialogs, and this one function has to become two in order to get the contents of the Windows handler functions right. If there is only one handler function in the Python code, one of the GUI elements will not be tied to this function by Python2C and the code inside this function will not be added to the Windows handler function for the dialog where this GUI element resides. If GUI element `gui1` calls function `handler_gui1` and `gui2` also calls the same function, then only `gui1` will get the translated code from `handler_gui1` added to its Windows handler function. A copy of `handler_gui1` has to made in this case, called `handler_gui2`. GUI element `gui2` needs to call this function instead to get the code inside `handler_gui2` added to its Windows handler function.

The Python code example found in chapter 2.6.1.4 illustrates the use of the naming convention. That example also illustrates quite well what sort of structuring of the Python code that is required to get the GUI translation to work with a one-pass translator. The example does not use the convention of the GUI class. Instead it uses markers in the code comments to indicate where the creation of all GUI elements begin and end. As seen in the example code these markers are "`# begin gui`" and "`# end gui`". Since all comments are ignored by the Python compiler and does not become a part of the Python AST, I had to abandon the idea of using the markers. The GUI class now serves the same purpose.

4.2.3 How to implement GUI translation as a part of Python2C

4.2.3.1 The "GUI translation mode"

To implement GUI translation as a part of Python2C the program would have to be able to tell a "regular" program from a program that includes a GUI. If a GUI is found as a part of the source code a "GUI translation mode" would have to be activated. "Regular" C++ code looks quite different from Windows CE GUI code. The GUI code has a `WinMain` function that must replace the `main` function, handles, resources and identifiers must be introduced, handler functions and callback functions must be added and the *resource script* has to be generated.

The "GUI translation mode" could be activated when Python2C comes across an `import` statement importing the `Tkinter` module. If the GUI mode is going to be triggered by finding this statement, it is required that it is put as early as possible in the source code. This mode would have to be active by the time the GUI class code is being traversed in the AST and the TST is being generated. Activating the mode as early as possible is the only thing that would ensure a correct translation of this class and also the important `WinMain` function.

Currently, Python2C's GUI translation mode is activated manually, though the "-g" option described in chapter 4.1.1.1. With a manual activation the mode will be active as Python2C scans the first line of the code. This certainly eliminates any problems or bugs that could arise as a result of having the mode triggered by elements in the source code. It is very likely that the person that is using Python2C to translate a program knows if the program supports a GUI or not. In this regard, manual activation works just as well as – or perhaps better than – triggering.

4.2.3.2 Additions to the data structures

For the GUI translation mode to work properly, a lot of the visitor functions would have to do a check to see if this mode has been activated. The same check would also have to be performed by many of the TST objects at the time of code translation, as the objects generate their part of the resulting C++ code. The new mode would have to generate a different TST than the one that would be generated otherwise. New classes of TST objects would have to be introduced or significant changes would have to be made to many of the existing classes.

The classes of TST nodes that hold the variables (`LangCreate`, `LangName` and `LangAssName`) will have to be amended to be able to hold information about variables holding GUI objects. Another, safer approach is not to alter any of the existing classes of TST nodes as it could introduce bugs in the translation of "non-GUI" code. When the GUI translation demands a change in the TST structure or different C++ code to be generated by certain TST objects, new objects that support these changes should be introduced as a part of the Python2C source code. `LangCreate` will turn into `LangGUICreate`, `LangName` will be replaced by `LangGUIName` and so on. Both the "regular" version and the GUI version of the TST objects will exist in the Python2C source code and the two versions would be used as necessary, while translating the same program. Changes and additions to the TST objects that hold the variables would also require changes in the Translator class. The `checkVariable` and `createVariable` functions would have to be amended, for instance.

The regular variables must be told apart from the variables that hold GUI objects. One way to pick them apart from the rest is to introduce a set of new variable types. Only a variable that holds a graphical component can have one of these types that will be linked to Python widgets and their function in the GUI, exclusively. A frame widget will be given the type "FRAME" internally in Python2C, a variable holding a button will be given the type "BUTTON" and so on. These types will not become a part of the translated C++ code. They are only meant to be an aid for Python2C as it is trying to determine what parts of the code relates to the GUI. With GUI types implemented, Python2C can more easily determine if a variable that is being operated upon holds a widget or not. Python2C can just check the type of a variable to determine if it holds a GUI object. In addition, it will also be able to tell exactly what kind of a function the widget has in the GUI, just by checking the type. Without the introduction of the GUI types Python2C could still find all of these things out by checking the symbol table, finding the object that the variable refers to and checking the contents of this object. This would be less efficient and would require more operations and lines of code.

4.2.4 To implement or not to implement?

That was the question. I decided that it would be to much work to do the former, so I did the latter. Implementing GUI translation would mean that I would have to rewrite and add to most of the existing visitor functions. Most of the TST object classes would also need to be revised and amended, and new classes of TST objects would have to be written. The lines of code would probably double to get some of it working and I can only guess at the increase of the number of bugs in the code. The Pocket PC 2002 operating system is already starting to get outdated and has been replaced by both Pocket PC 2003 and Pocket PC 2004. Creating a translation tool for an outdated system does not seem like a hot idea, although there are many similarities between the 2002 version and these new systems. There are also many similarities between GUI code for the Pocket PC and GUI code for the Windows XP operating system, for instance. I am still satisfied with figuring out how it can be done, even though I decided not to do it. Figuring this out was the aim of the thesis, not implementing a fully functional Python to C++ translator with support for GUI translation.

4.3 The problems of translation

A nice thing to know when working with the translation of programming languages is that it is always possible to translate the entire source code to the target language. It is just a matter of finding out how to do it and get around the obstacles that are in your way. The two languages can be used to create programs that do the same thing on the same type of computer. They may be different in syntax and in how they utilize the power of the hardware, but deep down they are essentially equivalent. This can be different when translating one human language to another. In this area you can frequently come across words or turns of phrase which are untranslatable: There does not exist equivalents in the target language.

This chapter describes some of the problems I have come across when writing Python2C, and it assumes that the reader has read the code documentation in chapter 4.1. Python2C's strengths and weaknesses are commented upon, and, in this regard, the table in chapter 4.1.2 serves as a good overview of what has been implemented in Python2C and what has not.

4.3.1 Variable creation

The Python syntax does not distinguish variable creation from just setting or changing the value stored in a variable. Variable creation and initialization is always done in one statement. You cannot create a variable without initializing it. If this statement contains a variable that has never been seen in the code before, this statement is a variable creation and initialization statement. If the variable has already been created and initialized this statement is just setting or changing the value that the variable holds.

One variable can hold a value of any type, and this type can also change at any time during execution, if the programmer so wishes. Types are never declared and cannot be declared as this is not a part of the Python syntax. Types do never the less exist internally in the Python interpreter. Each variable is automatically assigned a type depending on the value it holds. The programmer does not need to worry about it so much. Variable creation and initialization could look like this in C++: “`int i; i = 2;`”. Here it is divided into two separate statements. The first one creates the variable and gives it the type `int` without assigning a value to it. The second assigns the value, which is 2 in this case. In Python this has to be done in one simple statement: “`i = 2`”. This statement creates the variable `i` and gives it the type `int`, since the value being assigned to it is an integer, and sets the value to 2. If the variable has already been created earlier in the Python code, this statement will just change the value and/or the type. In C++ you could also write “`int i = 2;`”. This would take care of both the creation and the initialization in one statement.

4.3.1.1 Determining the types

To be able to translate variable creation and assignment from Python to C++ you first have to find a way to determine what type to assign to the variables. This means that you first have to figure out how to determine the type of the expression to the right of the “=”. When you have a statement like “`i = 2`”, the translator program just needs to check one constant. “2” is all that is to the left of the “=” in this case. 2 is an integer thus `i` gets the type `int`. Determining the type gets more complex if you have a statement like “`i = a.get()[1]`”. Now, the translator needs to break up the expression on the right. First, there is a variable `a` that probably holds an object that contains a function called `get`. The `get` function seems to return a list. The translator will have to determine the type of the elements stored in this list to be able to set the type of `i`. This may get more complicated if the object stored in `a` is an instance of a class that is defined in the source code that is being translated, as this part of the code may not have been looked at by the translator yet. If that is the case, there is no way that Python2C can come up with a correct type for `i`, and the code will have to be translated by a second traversal of the TST to create a working translation. Since Python2C currently is a one pass translator, it may fail to determine the type of such complex expressions quite frequently.

The algorithm for determining the type of a variable is contained in the `createVariable` function in the `Translator` class. This function attempts to determine the type of each new variable. It then creates a new variable object and adds this to the variable table. (For more on the variable table and the `createVariable` function see chapter 4.1.8.3.) All global variables have to be declared statically, outside of the `main` function in C++. Types of constants are determined by the `__init__` function in the `LangConst` class. The code of this function was included in the example in chapter 3.4.

The types themselves are stored as strings in the TST nodes. An integer variable can be represented as a `LangName` object with the type "INT". Python2C also supports the nesting of types as a type string can contain more than one type. If a variable holds a list of lists that contain integers, the type of this variable will be "LIST LIST INT", in that order. The string starts at the top of the "type hierarchy" and ends at the bottom. All the types that are supported by Python2C are listed in the `types` dictionary in the `Translator` class. This dictionary is found in chapter 4.1.8.5.

I have chosen to enforce the C++ rule that types of variables cannot change during execution, although they can do so in Python. Python2C will not translate a change of the type of a variable correctly, in its current version. To implement support for this in Python2C one could envision more than one possible solution. One solution could be to create a new variable for each time the type changes. If the integer variable `i` changes its type to `double`, Python2C could create a new `i_double` that is inserted into the translated C++ code wherever `i` occurs after the change of type. Another solution could be to create a variable `struct` or class of objects in C++. All variables would be an instance of this class or `struct`. It would provide storage for every possible type that a variable can have. With such a solution Python2C would not need to create a new variable for each time one variable changes its type. The contents of the `struct` or object would change during execution of the translated C++ program. These `struct`s or objects would look very similar to the `struct`s that are used to translate the Python lists, tuples and dictionary data types.

4.3.1.2 Has it already been created?

As already mentioned, variable creation and setting the value of a variable is all done in the same-looking statement in Python. You cannot tell from the syntax if the variable is being created when you find a statement like "`i = 2`" or if the value of i is being set. To find this out you have to look at the context where the statement occurs. According to the *most closely nested rule* ("[...] given several declarations for the same name, the declaration that applies to a reference is the one in the most closely nested block to the reference." [Louden, 1997]), this variable is a new variable if there is no other variable in the same block or in the above nested blocks named "`i`". If a variable `i` has been created before the above statement, then this statement is just setting the value of this variable. If `i` has not been created earlier in this block or in any above blocks, then it must be a statement creating the variable `i`.

Python2C uses the variable table to keep track of the variables that have been created. The variable table has a stack structure where each block in the program has its own stack element. The current block is the top element of the stack, and the previous blocks are found according to the most closely nested rule when going down the stack. Each time Python2C detects that the traversal of the AST enters a new block structure, a new stack element is pushed onto the variable table. When

Python2C leaves this block the element is popped from this stack. The bottom level of the stack is the global level, where you find all the static variables in C++. The `checkVariable` function in the `Translator` class is responsible for determining if a variable has previously been created or not. It checks with the variable table to see if it can find a variable with the same name as the variable it is currently checking. It starts by checking the top element of the variable table and goes down the stack from there. As soon as a matching name is found in one of the stack elements it is clear that the variable already exists, and the statement in question is just setting its value. If no matching name is found, the variable must be created and added to the variable table. In this case `checkVariable` calls `createVariable`, which is responsible for doing that.

4.3.1.3 Global variables and other special cases

The global code in a Python program does not reside inside any kind of block structures. All the variables that are created here are available by using the `global` keyword in Python. The `global` keyword is used to access global variables from parts of the Python code that reside inside a block structure, in other words; a class or a function. I have not implemented support for translation of the `global` keyword in Python2C, although this can be implemented fairly easily. As mentioned earlier, all the variables that have been created globally are found in the bottom element in the variable table in Python2C.

Python does not have a `main` function, like C++. The `main` function basically starts the program in C++, where the same thing is done by the statements in the global code in Python. In C++, global variables, forward declarations of functions and `#include` directives are more or less the only things that are found outside any block structure. You cannot have code statements in the global or static part of the code. In Python you can, as this global part of the code basically replaces the `main` function. This means that all the variables that are created globally in Python must be created statically in the translated C++ code. In other words; they must reside outside the `main` function. All the other statements that are a part of the global Python code must go inside the `main` function. Variables cannot even be given values in the static C++ code, they can only be created. Thus, a statement like "`i = 2`" will have to be divided into two C++ statements "`int i;`" and "`i = 2;`". The first one will be put in the static part of the C++ code and the second will have to reside inside the `main` function.

To make the translation of the global code a little easier I decided to create a class of TST objects called `LangMain`. The global level of the Python code is represented by a `Stmt` object in the Python AST. Normally, `Stmt` objects are replaced by `LangStmt` objects in the TST. I chose to make the global code a special case, as I needed to keep certain parts of the global code separate from each other. I wanted to store the code that goes inside the `main` function separate from the static variable declarations. This new class would also enable me to store all the forward declarations of the translated functions in one place. `LangMain` is a special version of the `LangStmt` class that can store all this extra information and keep it separate. It serves to represent the entire global part of the code in the TST and to translate this code to C++. Only one `LangMain` object can be present in any TST.

The example below illustrates the structuring that is being done by the `LangMain` object. As you can see, the forward declaration occurs before the rest of the program. In the case of multiple functions, all the forward declarations will be gathered after one another in that same location in the code. All the global variables are also gathered in one spot, above the `main` function. They are declared outside the `main` function and used inside of it. Thus, the original Python statements have been turned into two C++ statements, as described above.

Below is a simple Python program that takes a string and an integer as input. The integer sets the number of times the string shall be printed. A function does the actual printing.

```
#!/usr/bin/env python

def printOut(number, string):
    s = ""
    for i in range(number):
        if(i % 5) == 0:
            print "\n" + string
        else:
            print string

string = raw_input("Type a string: ")
temp_string = raw_input("and the number of times it should be printed: ")
number = int(temp_string)
printOut(number, string)
print "\nString \"" + string + "\" printed " + str(number) + " times."
```

This is the C++ translation of the above Python program, as produced by Python2C.

```
/* File translated Wed Aug 24 14:35:51 2005
 * by Python2C version 0.012. */

#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>

/* The python types translation headers: */
#include "python2clist.h"
#include "python2ctuple.h"
#include "python2cdict.h"

/* Forward declarations: */
void printOut(int number, char* string);

void printOut(int number, char* string)
{
  char* s;
  int i;
  s = (char*) malloc(256);
  s = "";
  for(i = 0; i < number; i += 1)
    {
      if(i % 5 == 0)
      {
        cout << "\n" << string << endl;
      }
      else
      {
        cout << string << endl;
      }
    }
}
```

```
char* string;
char* temp_string;
int number;

int main(int argc, char* argv[])
{
  printf("Type a string: ");
  string = (char*) malloc(256);
  scanf("%s", string);
  printf("and the number of times it should be printed: ");
  temp_string = (char*) malloc(256);
  scanf("%s", temp_string);
  number = atoi(temp_string);
  printOut(number, string);
  cout << "\nString \"" << string << "\" printed " << number << " times." <<
endl;
  return 0;
}
```

The `createVariable` function in the `Translator` class determines if a variable is global or not. This is done with the help of the flags that are located in the VisitorFunctions class. These flags are `inside_class`, `inside_function` and `inside_cond`. These flags indicate whether the code that is currently being traversed in the AST is located inside a class, function or an if, else, for or while block respectively. If the current code is global, all the flags will be set to 0. If the code is located inside a for loop inside a function inside a class, all the flags will be set to 1. The flags are set by the visitor functions responsible for traversing the relevant parts of the code. The `visitClass`'s functions is responsible for setting the `inside_class` flag to 1 when the traversal of the AST enters the code body of a class. It is also responsible for setting it back to 0, when the traversal of the class code has finished. In the same way the other flags are set by the relevant visitor functions.

Before setting a flag, a check needs to be done for whether the flag was already set. This can occur when you have functions inside of functions or nested loops, for instance. When this is the case, it is very important that one of the nested blocks does not set the relevant flag to 0 before it is time to do so. Only the outermost of the nested blocks should set this flag to 0.

4.3.2 Functions

When implementing the support for translating Python functions to C++ functions, one realizes that determining the types proves to be the most difficult problem. Python is a language that is not particularly concerned with the types of anything, while C++ certainly is. Both the argument variables to functions and the return value from functions do not have fixed types in Python, as they must have in C++. A Python function can basically accept any type of argument and return any type of value as a result. This does require that the “inner workings” of the function supports this. Many operations can only be done on certain types of data in Python, as with any other programming language. When trying to determine the types of both the arguments to and the return value from a function, the translator program will have something to go on here. The code inside the function will tell you something about what types of data this function is meant to accept and return. The values that are passed to the function from other parts of the code will also provide information about the types of the argument variables.

In C++, the return value from a function can only be of one fixed type. The same is the case for the argument variables. Python2C will have to try to determine the types of all arguments and returns for each function in the Python code. This requires that the programmer of the Python program that is being translated, takes care to program functions that only accept and return the same types of data each time. Anything else might lead to conflicting types and errors. If the programmer needs to have more than one type of data returned by one function, another function should be written that takes care of the other type of data. This is what you would have to do in C++. One could of course design a translator that creates multiple C++ functions from one Python function, when necessary. I have not invested any work in trying to do that, as it would take up a lot of time.

4.3.2.1 Determining the return type

The return type of a function is currently set by the `visitReturn` function in Python2C. This is done by a call to the function `appendReturnType` in the `LangFunction` object. This function appends the return type to a list of return types, and the final return type is determined on the basis of what types exist in this list. When the traversal of the entire function definition is done the `determineReturnType` function in the `LangFunction` object is called to do this. Many functions will contain more than one return statement, and all of these statements will add its type to the return types list. This ensures that all return statements are taken into consideration to produce the final return type from the function. If no return statement is found in the code of the function, the function's return type is set to `void`.

If it is clear from the code contained in the function what the type of the return is, this way of determining the type works well. In some cases it is not clear what the type of the return might be, because the operations that is being done inside the function might support more than one type. Sometimes the return type of the function relies on what the types of the argument variables are. In such cases, the types of these variables have to be determined before the return type, to be able to get the translation right. In many cases this requires more than one pass of the code.

This is a small example of how a very simple Python function can be impossible to translate without creating more than one C++ function. The `add` *function in the Python program below can add any two argument variables passed to it and return the result. The types of the arguments do not matter as long they can be added, otherwise the program will crash.*

```
#!/usr/bin/env python

def add(arg1, arg2):
    return arg1 + arg2

i = add(2, 2)
print "add returned " + str(i) + "."
list1 = [1, 2]
list2 = [3, 4]
list = add(list1, list2)
print "add returned " + str(list) + "."
```

This is the translation of the above code as it is produced by Python2C. As you can clearly see, the passing of both integers and lists to the `add` *function leads to confusion about the type of its return* *value. The return type of the function is never set. This, in its turn, leads to further errors. With no determined return type for the `add` function, it is impossible for Python2C to determine the type of variables `i` and `list`. The types of both of these variables depend on the return type of the `add` function.*

```
/* File translated Fri Aug 26 13:32:50 2005
 * by Python2C version 0.013. */

#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>

/* The python types translation headers: */
#include "python2clist.h"
#include "python2ctuple.h"
#include "python2cdict.h"

/* Forward declarations: */
/* Python2C Error NOTYPE */ add(int arg1, int arg2);

/* Python2C Error NOTYPE */ add(int arg1, int arg2)
{
  return arg1 + arg2;
}

/* Python2C Error NOTYPE */ i;
struct List* list1;
struct List* list2;
/* Python2C Error NOTYPE */ list;

int main(int argc, char* argv[])
{
  i = add(2, 2);
  cout << "add returned " << i << "." << endl;
  list1 = createIntList();
  appendList(list1, 1);
  appendList(list1, 2);
  list2 = createIntList();
  appendList(list2, 3);
  appendList(list2, 4);
  list = add(list1, list2);
  cout << "add returned " << list << "." << endl;
  return 0;
}
```

4.3.2.2 Determining the types of argument variables

Python2C is currently determining the types of the argument variables on the basis of what is being passed as arguments to the functions through function calls. If the type of a variable that is being passed as an argument has been determined, the type of the corresponding function argument will be set. If the type of the variable that is being passed has not yet been determined, the type of the function argument will not be determined at this time. It might be determined by another call to the same function, later on. The types of the argument variables are passed to the function object in the function table by the `visitCallFunc` function. There is very limited support for complex expressions being passed as arguments. Passing constants and variables as arguments will work fine in most cases in the current version of Python2C. I have not spent time implementing any support for anything more complex than this.

Argument variables can also, to some extent, be determined by they way they are used inside the code of the function, similar to the way that the return type is determined. By having Python2C checking this as well, one would be able to determine the types of the arguments even more accurately. This remains to be implemented, however.

4.3.2.3 Forward declarations and other issues

A forward declaration is created by the `createForward` function in the `LangFunction` object. As soon as the code of the function has been traversed the forward declaration can be generated correctly. This function is called by `visitCallFunc` which checks the function table for the definition of the function that is being called. If this function turns out to be global, a forward declaration should be created and added to the `LangMain` object. `visitCallFunc` does exactly that. `createForward` is also called by the `LangClass` object at the time of code translation. The `LangClass` object needs the forward declaration to be able to generate the C++ class definition correctly. This definition contains forward declarations of each function inside the class.

Functions can exist inside of functions, but I have not put any effort into translating these things. Nested functions in the Python source code is sure to produce undesired behavior in Python2C in its current version.

4.3.3 Loops and `if` statements

I have loosely labeled loops and if statements as "conditionals" in the code of Python2C and in the code comments. They all have a condition for whether a certain piece of code should be executed or not. They are also code blocks that can be nested within each other. I variable that is created inside one of these blocks will not exist outside the block. It is in use for that limited time only, as is also the case with functions. Variables inside classes can however be reached from the code residing outside of the class' block. The `inside_cond` flag in the `VisitorFunctions` class is used to tell the translation engine if the current code is located inside a conditional block. These blocks include `if`, `else`, `elif`, `for` and `while` structures.

The `else` and `elif` code blocks are parts of the `if` structure in Python. Each `if` statement starts with the `if` keyword that marks the beginning of the `if` block. The `elif` or `else` block follows the `if` block, although these blocks are not required to form a syntactically correct `if` statement. For more on the syntax of `if` statements consult the Python documentation from Python.org.

The syntax of the `if` statement is quite similar in Python and C++. Only trivial changes has to be made to the syntax to produce a working translation, like the addition of parenthesis and brackets, for instance. This is also the case with the `while` statement. One thing that needs special attention is the `inside_cond` flag. This flag needs to be set at the right time to produce a correct translation and to add any new variable that is created inside these blocks to the variable table correctly.

4.3.3.1 The for statement

The syntax of the `for` statements do differ when comparing Python to C++. The main difference is the fact that C++ must have a counter variable as a part of its `for` statement, and Python does not. C++ normally uses an integer variable that is incremented or decremented by each repetition of the loop. When the this counter variable reaches a certain value, the loop is broken. A test is performed by each repetition to check if the variable has reached this value or not. This test is also a part of the `for` statement in C++.

Python's `for` loops are “list oriented”. The `for` loops are specially designed to loop though a Python list from beginning to end or from a start index to a stop index. The syntax is basically as follows in the for statement: “`for i in list:`”. The variable `i` will always contain a copy of what is stored at the list's current index. It is not a counter variable. The body of the `for` loop block follows the “:”. In this case, the `for` loop will loop though the list from beginning to end. To loop through the list from a start index to a stop index the syntax is as follows: “`for i in list[start:stop]:`”. The Python `range` function is often used to produce a list of integers that can be used as counters almost like the counter variables are used in C++. Such a `for` statement would look something like this: “`for i in range(start, stop, increment):`”. The variable `i` can now be used as an index to look up elements in another list, for instance.

The differences in Python and C++ syntax usually means that new variables have to be introduced in the target code that were not originally a part of the source code. How these new variables are created is explained in chapter 4.1.8.3. New variables cannot be introduced inside a `for` statement in C++ and the counter variables must be created outside the `for` block in the translated code. The code below shows how the translation of the various Python `for` loop constructions are translated to C++, in detail.

This is a small and simple Python program that loops though the arguments in the `sys.argv` list.

```
#!/usr/bin/env python

import sys, math

print "Hello, World!"
for x in sys.argv[1:]:  # Loop through the list of arguments, compute the sin-
  r = float(x)          # value and print the results.
  s = math.sin(r)
  print "sin(" + str(r) + ")=" + str(s)
```

This is the C++ translation of the above Python program as it is produced by Python2C.

```
/* File translated Mon Aug 29 18:11:31 2005
 * by Python2C version 0.013. */

#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>

/* The python types translation headers: */
#include "python2clist.h"
#include "python2ctuple.h"
#include "python2cdict.h"
#include <math.h>
```

```
char* x;
int __py2c__for_counter;
int __py2c__for_length;

int main(int argc, char* argv[])
{
  __py2c__for_length = argc;
  cout << "Hello, World!" << endl;
  x = (char*) malloc(256);
  for(__py2c__for_counter = 1; __py2c__for_counter < __py2c__for_length;
__py2c__for_counter++)
    {
      double r;
      double s;
      x = argv[__py2c__for_counter];
      r = atof(x);
      s = sin(r);
      cout << "sin(" << r << ")=" << s << endl;
    }
  return 0;
}
```

In the C++ translation above the number of lines of code has increased significantly. Python2C has introduced two new variables: `__py2c__for_counter` and `__py2c__for_length`. The first one of these are used as the loop counter, the second is used to store the length of the list that is being looped through. Also notice that the Python `sys.argv` is translated to the C++ `argv` argument to the `main` function. Another alternative would have been to translate `sys.argv` to an actual list structure in C++ and not use the `argv` array. All other Python list structures are translated to list in C++. The `sys.argv` is the only exception.

Below is a simple Python program that demonstrates the use of the `range` function in a `for` loop.

```
#!/usr/bin/env python

import sys, math

s = "Hello, World!"
length = len(s)
for i in range(0, length, 1):
  print str(i) + " " + s[i]
```

This is the C++ translation of the above Python program as it is produced by Python2C. In this case no new variables are created. The `length` argument variable to the `range` function is used in the test, and the variable `i` is used as the counter. Both the variables must be created outside the `for` block in C++, and since the `for` block is global in the Python code, both the variables must be created outside the `main` function in C++.

```
/* File translated Mon Aug 29 18:02:33 2005
 * by Python2C version 0.013. */
```

```
#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>

/* The python types translation headers: */
#include "python2clist.h"
#include "python2ctuple.h"
#include "python2cdict.h"
#include <math.h>

char* s;
int length;
int i;

int main(int argc, char* argv[])
{
  s = (char*) malloc(256);
  s = "Hello, World!";
  length = strlen(s);
  for(i = 0; i < length; i += 1)
    {
      cout << i << " " << s[i] << endl;
    }
  return 0;
}
```

The C++ `argv` array is being looped through in both the above translation examples. It is of course possible to do the same thing with actual list structures, although support for this has not been fully implemented in Python2C. Some looping though actual lists is supported, however. The next chapter contains an example of this.

4.3.4 Lists, tuples and dictionaries

As mentioned, all Python lists are translated into actual lists in C++ with the exception of `sys.argv`. Python tuples and dictionaries are also translated into C++ equivalents. Lists, tuples and dictionaries do not exists as C++ data types, so I had to create these myself. How this was done has already been described in detail in chapter 4.1.8.6. How these data types are used when translating from Python to C++ is best explained through the use of some code examples.

This is a simple Python program that exemplifies most of the list operations that is currently supported by Python2C.

```
#!/usr/bin/env python

list1 = [1, 2, 3, 4, 5, 6]
list2 = [7, 8, 9]

# list becomes a copy of the two lists.
list = list1 + list2

print "list  = " + str(list)
print "list1 = " + str(list1)
print "list2 = " + str(list2)
```

```
list.append(10)
list.append(11)

print "list     = " + str(list)
print "list[10] = " + str(list[10])

list2 = list1
print "list2 = " + str(list2)

del list1[5]
print "list1 = " + str(list1)
print "list2 = " + str(list2)

for i in range(len(list)):
    print "list[" + str(i) + "] = " + str(list[i])

i = list[3]
print "i = " + str(i)
list[1] = list[0] = 999
print "list[0] = " + str(list[0]) + ", list[1] = " + str(list[1])
print "list = " + str(list)
```

Here is the C++ version of the above Python program as translated by Python2C.

```
/* File translated Mon Jan 10 15:26:18 2005
 * by Python2C version 0.009. */

#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>

/* The python types translation headers: */
#include "python2clist.h"
#include "python2ctuple.h"
#include "python2cdict.h"

struct List* list1;
struct List* list2;
struct List* list;
struct ListNode* __py2c__subscript_temp_var0;
int i;
struct ListNode* __py2c__subscript_temp_var2;
struct ListNode* __py2c__subscript_temp_var3;
struct ListNode* __py2c__subscript_temp_var4;

int main(int argc, char* argv[])
{
  list1 = createIntList();
  appendList(list1, 1);
  appendList(list1, 2);
  appendList(list1, 3);
  appendList(list1, 4);
  appendList(list1, 5);
  appendList(list1, 6);
  list2 = createIntList();
  appendList(list2, 7);
  appendList(list2, 8);
  appendList(list2, 9);
  list = createList();
  list = addLists(list, list1);
  list = addLists(list, list2);
  cout << "list  = " << printList(list) << endl;
  cout << "list1 = " << printList(list1) << endl;
  cout << "list2 = " << printList(list2) << endl;
  appendList(list, 10);
  appendList(list, 11);
  cout << "list     = " << printList(list) << endl;
  __py2c__subscript_temp_var0 = getListNode(list, 10);
  cout << "list[10] = " << printListNode(__py2c__subscript_temp_var0) << endl;
  list2 = list1;
  cout << "list2 = " << printList(list2) << endl;
  deleteListNode(list1, 5);
  cout << "list1 = " << printList(list1) << endl;
  cout << "list2 = " << printList(list2) << endl;
  for(i = 0; i < list->length; i += 1)
    {
      struct ListNode* __py2c__subscript_temp_var1;
      __py2c__subscript_temp_var1 = getListNode(list, i);
                      cout    <<    "list["    <<    i    <<    "]    =    "    <<
printListNode(__py2c__subscript_temp_var1) << endl;
    }
  __py2c__subscript_temp_var2 = getListNode(list, 3);
  i = __py2c__subscript_temp_var2->int_value;
  cout << "i = " << i << endl;
  setListValue(list, 1, 999);
  setListValue(list, 0, 999);
  __py2c__subscript_temp_var3 = getListNode(list, 0);
  __py2c__subscript_temp_var4 = getListNode(list, 1);
   cout  <<  "list[0]  =  "  <<  printListNode(__py2c__subscript_temp_var3)  <<  ",
list[1] = " << printListNode(__py2c__subscript_temp_var4) << endl;
  cout << "list = " << printList(list) << endl;
  return 0;
}
```

In the above example the Python statements creating `list1` and `list2` have been translated to a call to `createIntList` and a series of calls to `appendList`. `createIntList` creates the list `stuct` and sets the type to `IntType`. There are separate functions for creating lists that contain doubles, strings, lists and so on. `appendList` adds the `ListNode struct`s to the list that is passed as the first argument. The second argument to this function is the value. The `list` pointer is set by a call to `createList`, as it is unclear what type this list is going to hold, at the point of creating it. The type of a list that is created like this will be set when the first element is appended or when the pointer is set to point to another list. Until then, the type of `list` will be `NoType`. Deletion is translated by a call to `deleteListNode` and the printing is being done by printing the strings returned by the calls to the `printList` function. A number of temporary variables, that are not present in the Python code, are created as necessary.

Python tuples do not support some of the operations that can be performed on a Python list. You cannot append an element to a Python tuple and you cannot delete an element from a Python tuple. You can change the contents of a Python tuple by setting it to point to another tuple or adding tuples together. It is also possible to set or change the value stored in a tuple element, although this is not being translated by Python2C at the moment. This remains to be implemented.

This is a simple Python program that exemplifies most of the tuple operations that is currently supported by Python2C.

```
#!/usr/bin/env python

tuple1 = (1, 2, 3, 4, 5, 6)
tuple2 = (7, 8, 9)

# tuple becomes a copy of the two tuples.
tuple = tuple1 + tuple2

print "tuple  = " + str(tuple)
print "tuple1 = " + str(tuple1)
print "tuple2 = " + str(tuple2)

tuple2 = tuple1
print "tuple2 = " + str(tuple2)

for i in range(len(tuple)):
    print "tuple[" + str(i) + "] = " + str(tuple[i])

print "tuple[0] = " + str(tuple[0]) + ", tuple[1] = " + str(tuple[1])
print "tuple = " + str(tuple)
```

This is the C++ translation of the above Python program as produced by Python2C.

```
/* File translated Mon Jan 10 15:48:57 2005
 * by Python2C version 0.009. */

#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>

/* The python types translation headers: */
#include "python2clist.h"
#include "python2ctuple.h"
#include "python2cdict.h"

struct Tuple* tuple1;
struct Tuple* tuple2;
struct Tuple* tuple;
int i;
struct TupleNode* __py2c__subscript_temp_var1;
struct TupleNode* __py2c__subscript_temp_var2;

int main(int argc, char* argv[])
{
  tuple1 = createTuple();
  appendTuple(tuple1, 1);
  appendTuple(tuple1, 2);
  appendTuple(tuple1, 3);
  appendTuple(tuple1, 4);
  appendTuple(tuple1, 5);
  appendTuple(tuple1, 6);
  tuple2 = createTuple();
  appendTuple(tuple2, 7);
  appendTuple(tuple2, 8);
  appendTuple(tuple2, 9);
  tuple = createTuple();
  tuple = addTuples(tuple, tuple1);
  tuple = addTuples(tuple, tuple2);
  cout << "tuple  = " << printTuple(tuple) << endl;
  cout << "tuple1 = " << printTuple(tuple1) << endl;
  cout << "tuple2 = " << printTuple(tuple2) << endl;
  tuple2 = tuple1;
  cout << "tuple2 = " << printTuple(tuple2) << endl;
  for(i = 0; i < tuple->length; i += 1)
    {
      struct TupleNode* __py2c__subscript_temp_var0;
      __py2c__subscript_temp_var0 = getTupleNode(tuple, i);
                      cout    <<    "tuple["    <<    i    <<    "]    =    "    <<
printTupleNode(__py2c__subscript_temp_var0) << endl;
    }
  __py2c__subscript_temp_var1 = getTupleNode(tuple, 0);
  __py2c__subscript_temp_var2 = getTupleNode(tuple, 1);
   cout  <<  "tuple[0]  =  "  <<  printTupleNode(__py2c__subscript_temp_var1)  <<  ",
tuple[1] = " << printTupleNode(__py2c__subscript_temp_var2) << endl;
  cout << "tuple = " << printTuple(tuple) << endl;
  return 0;
}
```

As seen above, the translation for the supported tuple operations is done in the same way as with the translation of operations on lists. The only difference is that the C tuples have their own functions. Where `createList` was being used in the previous example with lists, this function has been replaced by `createTuple`. `appendList` has been replaced by `appendTuple`, and so on. (The C tuples support appending although the Python tuples do not.) Tuples can contain a collection of values that have different types, thus there are no need for functions that create tuples with different types. The `Tuple struct` does not have a type, but the values stored in the `TupleNodes` certainly do.

The Python dictionary does not support the addition of two dictionaries and there exists no append function for the Python dictionary. There is no need for such a function since a dictionary is not a sequential data structure where the indexes are numbered. This data structure is indexed by strings that are called *keys*. A new element is inserted into a dictionary by setting the value at the string index. This operation is currently not being translated by Python2C. Looping though the elements of a dictionary is not supported either. There are also some bugs in the source code for the C dictionary that remain to be fixed.

This is a simple Python program that exemplifies most of the dictionary operations that is currently supported by Python2C.

```
#!/usr/bin/env python

dict1 = {"jalla"  : 1,
         "sommer" : 2,
         "tusen"  : 3,
         "mega"   : 4,
         "ransel" : 5,
         "kamel"  : 6}
```

```
dict2 = {"smalahovud" : 7,
         "saklig"     : 8,
         "nabo"       : 9}

print "dict1 = " + str(dict1)
print "dict2 = " + str(dict2)

dict2 = dict1
print "dict2 = " + str(dict2)

del dict1["jalla"]
print "dict1 = " + str(dict1)
print "dict2 = " + str(dict2)

i = len(dict1)
print "len(dict1) = " + str(i)
```

This is the C++ translation of the above Python program as output by Python2C. The translation of dictionaries and the operations on them is done much in the same way as when translating lists or tuples. Dictionaries have types as only one type of data can be stored in one dictionary at one time.

```
/* File translated Mon Aug 29 20:56:36 2005
 * by Python2C version 0.013. */

#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>

/* The python types translation headers: */
#include "python2clist.h"
#include "python2ctuple.h"
#include "python2cdict.h"

struct Dict* dict1;
struct Dict* dict2;
int i;

int main(int argc, char* argv[])
{
  dict1 = createDictOfType(IntType);
  dictInsert(dict1, "jalla", 1);
  dictInsert(dict1, "sommer", 2);
  dictInsert(dict1, "tusen", 3);
  dictInsert(dict1, "mega", 4);
  dictInsert(dict1, "ransel", 5);
  dictInsert(dict1, "kamel", 6);
  dict2 = createDictOfType(IntType);
  dictInsert(dict2, "smalahovud", 7);
  dictInsert(dict2, "saklig", 8);
  dictInsert(dict2, "nabo", 9);
  cout << "dict1 = " << dict1 << endl;
  cout << "dict2 = " << dict2 << endl;
  dict2 = dict1;
  cout << "dict2 = " << dict2 << endl;
  deleteDictItem(dict1, "jalla");
  cout << "dict1 = " << dict1 << endl;
  cout << "dict2 = " << dict2 << endl;
  i = dict1->num_elements;
  cout << "len(dict1) = " << i << endl;
  return 0;
}
```

4.3.5 Classes and objects

The Python syntax for writing classes is quite simple and is exemplified through the code examples found in this chapter. The C++ syntax differs quite a bit from the simple set-up that is found in Python. Code for C++ classes are usually put in separate files that define a class interface and the functions that belong to the class. Memory management issues also do their part to complicate the use of classes in C++, and the Python counterparts come out looking quite streamlined and flexible in comparison. The translation of classes is probably best explained through the use of code examples.

In the example below, a small Python program containing one simple class is translated to C++ by Python2C. As usual, the number of lines of code increases significantly in the target code. Translations of code that contains one or more class definitions will produce one file that contains the main program and one file for each class definition. The class definition files contain a C++ class interface. This interface contains forward declarations of the class' functions and holds its data, variables and pointers. Everything listed in the interface can be declared as `public` or `protected`. Since Python does not have any protection of class variables or functions, all of these will be translated as `public` in every case.

Python2C adds two functions to the C++ class in the translation below, that were not present in the original source code. These are the constructor and destructor functions, called `Print` and `~Print` in this case. A Python class definition does not have to include an explicit constructor, but a C++ class definition must have one. This is why an empty constructor is added to the translated C++ class file. If the constructor function is present in the Python code, its code will of course be translated and put inside the translated constructor. The destructor does not have to be present to make the program work, it is just considered to be a good programming practice. The code body of the destructor is not generated by Python2C and must be written manually.

This is an example of a Python program using a small and simple class that contains one function and no constructor.

```
#!/usr/bin/env python

class Print:
    def printOut(self, number, string):
        s = ""
        for i in range(number):
            if(i % 5) == 0:
                print "\n" + string
            else:
                print string

string = raw_input("Type a string: ")
temp_string = raw_input("and the number of times it should be printed: ")
number = int(temp_string)
```

```
print_object = Print()
print_object.printOut(number, string)
print "\nString \"" + string + "\" printed " + str(number) + " times."
```

This is one of the files that make up the translation of the above Python program. This particular program produces two files that hold the translated code. The first one is the code that holds the main function, and the second is the file containing the class definition. The class definition file is imported through an `#include` statement in the file containing the main code.

```
/* File translated Sun Nov 14 17:57:57 2004
 * by Python2C version 0.006. */

#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>
#include "Print.h"

char* string;
char* temp_string;
int number;
Print* print_object;

int main(int argc, char* argv[])
{
  printf("Type a string: ");
  string = (char*) malloc(256);
  scanf("%s", string);
  printf("and the number of times it should be printed: ");
  temp_string = (char*) malloc(256);
  scanf("%s", temp_string);
  number = atoi(temp_string);
  *print_object = Print();
  print_object->printOut(number, string);
   cout << "\nString \"" << string << "\" printed " << number << " times." <<
endl;
  return 0;
}
```

This is the file `Print.h` containing the class definition.

```
/* File translated Sun Nov 14 17:57:57 2004
 * by Python2C version 0.006. */

#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>

class Print
{
 public:
  ~Print();
  void printOut(int number, char* string);
  Print();
};

void Print::printOut(int number, char* string)
{
  char* s;
  int i;
  s = (char*) malloc(256);
  s = "";
  for(i = 0; i < number; i += 1)
    {
      if(i % 5 == 0)
      {
        cout << "\n" << string << endl;
      }
      else
      {
        cout << string << endl;
      }
    }
}

Print::Print()
{
}

Print::~Print()
{
}
```

Below is an example of a Python program that uses two classes. The main difference between the two classes is that one of them has a defined constructor function, the `__init__` function, and the other does not. Both the classes have variables that are stored in the instance objects, although these are never actually used for anything. Python2C has no real support for statements containing `AssAttr` expressions. This means that Python2C will not translate expressions like `self.unused_var1` correctly. This requires the ability to understand what the variable `self` holds or points to. That, in its turn, requires the ability to trace the contents of `self` back to its origin. Currently, this has not been implemented.

As with the previous example, there is a class named `Print`. This time the definition if the `__init__` function causes the translation of the code body to be put inside the C++ constructor `Print`. In the previous example this function remained empty. The second class, named `PrintSomeMore`, has no definition of the constructor in the source code but still gets a translation that has a C++ constructor with a statement in the code body. This is because the variable `unused_var1` cannot be initialized in the class interface, as this is not allowed in C++. The value is set to 0 in the constructor instead. `#include` directives are generated for both of the class files. The main code file includes both the files and the class files include each other by default. The fact that the class files include each other may lead to errors when compiling the translated code. The files will include themselves in an infinite loop. To fix this problem you have to remove the `#include "Print.h"` directive in the `PrintSomeMore.h` file and the `#include "PrintSomeMore.h"` in the `Print.h` file. Another bug in Python2C is the fact that all variables that are declared in the body of the class code are added to the interface of each class. These bits of code also has to be edited by hand. Python's `len` function is not translated correctly either.

This is a Python program that uses two classes to do some trivial printing.

```
#!/usr/bin/env python

class Print:
    unused_var1 = ""
    unused_var2 = 2

    def __init__(self):
        print "Object created!"

    def printOut(self, number, string):
        for i in range(number):
            if(i % 5) == 0:
                print "\n" + string
            else:
                print string

class PrintSomeMore:
   unused_var1 = 0

   def printOut(self, number, string):
       if number > len(string):
           number = len(string)
       for i in range(number):
           print string[i]

string = raw_input("Type a string: ")
temp_string = raw_input("and the number of times it should be printed: ")
number = int(temp_string)
print_object = Print()
print_object.printOut(number, string)
print "\nString \"" + string + "\" printed " + str(number) + " times."

string = raw_input("Type another string: ")
temp_string = raw_input("and the number of characters you want printed from this
string: ")
number = int(temp_string)
printsomemore_object = PrintSomeMore()
printsomemore_object.printOut(number, string)
print "\nThe first " + str(number) + " characters of \"" + string + "\" was
printed."
```

This is the contents of the translated file containing the main function.

```
/* File translated Wed Aug 31 13:37:54 2005
 * by Python2C version 0.013. */

#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>

/* The python types translation headers: */
#include "python2clist.h"
#include "python2ctuple.h"
#include "python2cdict.h"
```

```
/* The includes generated by the translation: */
#include "Print.h"
#include "PrintSomeMore.h"

char* string;
char* temp_string;
int number;
Print* print_object;
PrintSomeMore* printsomemore_object;

int main(int argc, char* argv[])
{
  printf("Type a string: ");
  string = (char*) malloc(256);
  scanf("%s", string);
  printf("and the number of times it should be printed: ");
  temp_string = (char*) malloc(256);
  scanf("%s", temp_string);
  number = atoi(temp_string);
  *print_object = Print();
  print_object->printOut(number, string);
  cout << "\nString \"" << string << "\" printed " << number << " times." <<
endl;
  printf("Type another string: ");
  scanf("%s", string);
  printf("and the number of characters you want printed from this string: ");
  scanf("%s", temp_string);
  number = atoi(temp_string);
  *printsomemore_object = PrintSomeMore();
  printsomemore_object->printOut(number, string);
  cout << "\nThe first " << number << " characters of \"" << string << "\" was
printed." << endl;
  return 0;
}
```

This is the translated `Print.h` *file.*

```
/* File translated Wed Aug 31 13:37:54 2005
 * by Python2C version 0.013. */

#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>

/* The python types translation headers: */
#include "python2clist.h"
#include "python2ctuple.h"
#include "python2cdict.h"
#include "PrintSomeMore.h"

class Print
{
 public:
  char* unused_var1;
  int unused_var2;
  int unused_var1;
  ~Print();
  void printOut(int number, char* string);
  Print();
};

Print::Print()
{
  unused_var2 = 2;
  unused_var1 = "";
  unused_var1 = (char*) malloc(256);
  cout << "Object created!" << endl;
}

void Print::printOut(int number, char* string)
{
  int i;
  for(i = 0; i < number; i += 1)
    {
      if(i % 5 == 0)
      {
        cout << "\n" << string << endl;
      }
      else
      {
        cout << string << endl;
      }
    }
}

Print::~Print()
{
}
```

This is the translated `PrintSomeMore.h` *file.*

```
/* File translated Wed Aug 31 13:37:54 2005
 * by Python2C version 0.013. */

#include <stdlib.h>
#include <stdio.h>
#include <iostream.h>
#include <fstream.h>

/* The python types translation headers: */
#include "python2clist.h"
#include "python2ctuple.h"
#include "python2cdict.h"
#include "Print.h"

class PrintSomeMore
{
 public:
  char* unused_var1;
  int unused_var2;
  int unused_var1;
  ~PrintSomeMore();
  void printOut(int number, char* string);
  PrintSomeMore();
};
```

```
void PrintSomeMore::printOut(int number, char* string)
{
  int i;
  if(number > len(string))
    {
      number = len(string);
    }
  for(i = 0; i < number; i += 1)
    {
      cout <<  << endl;
    }
}

PrintSomeMore::PrintSomeMore()
{
  unused_var1 = 0;
}

PrintSomeMore::~PrintSomeMore()
{
}
```

Each `LangClass` object holds the name and the path to the file where the translated class code will end up. These objects' `getString` function returns the empty string and writes the translated code to this file on its own. The code that resides outside of classes is written to file by the `writeCode` function in the `Translator` class. Each new `LangClass` object also adds a new type to the types table in the `Translator` class and gives it the same name as the name of the class with the string “`POINTER `” in front. Any variable or expression in the source code that is found to point to an object of this class will be given this pointer type by Python2C. Its equivalent in the C++ translation is a regular pointer to a class object. `Print* print_object` from the C++ translation example above, is such a pointer. The `Translator` class' types table will hold a mapping between the internal `POINTER Print` type and the C++ `Print*` in this case.

The constructor function of any class will be given the type “`OBJECT `“ followed by the class name. This type signals that the constructor actually returns an object and not a pointer to an object, an important thing to get right to make the translation work. All new types are added to the types table in the `Translator` class, and this is no exception. As seen in the above example, the statement `*print_object = Print();` contains a cast. The object returned by the `Print` constructor is pointed to by `print_object`. All such statements are translated in this way by Python2C; a class object is always pointed to and never held by a variable in the target code.

5 Project evaluation

Through working on this thesis I feel that I have covered a lot of ground. I have studied the relevant languages and hardware systems and the inner workings of compilers in theory and practice. I studied other attempts at creating translator programs and investigated a few alternative approaches to creating this kind of software.

The time that was set aside for all this work was limited and it was never my intention to produce a fully functional and optimal translator software bundle, that would be able to completely translate the Python language to C++. I had to make priorities and decide what parts of my program that needed the focus of attention. A lot of wanted functionality was sacrificed because of this time shortage, but this can always be implemented at some later point, if someone should wish to pick the work up where I left off. I invested most of the labor into translating Python to “standard” C++ and getting most of the basic language elements translated. Once I figured out how to do this, in a relatively good manner, the remaining functionality could be implemented in much the same way. I never had time enough to get around to implementing the Windows CE GUI translation, but I have explained how this can be done. The translation engine has been created, it just needs tweaking and a few upgrades to become more powerful. Python2C is capable of translating GUIs if someone takes the time to do this.

One could argue that I should have implemented multiple passes of the TST in order to get more of the translation working and to increase support for any type of valid and syntactically correct Python program. It would also minimize the need for new programming styles or conventions that I have suggested, especially related to the translation of GUIs. I see this as being outside the scope of this thesis, yet there is no doubt in my mind that a multiple pass translator would be more powerful. A complete translator software package would also include a target code optimizer. I have chosen not to look further into that problem either, since it would just take up too much time.

6 Appendix

6.1 The source code for the modified version of Strout and Heise's Python2C

This is the complete source code for the modified version of Strout and Heise's Python2C that I created.

```
#!/usr/bin/env python

#----------------------------------------------------------------------
# Python2C        Tom Simonsen
#
# Changes made to the original Strout/Heise code:
#     - The deprecated regex module changed to the re module.
#       Regex expressions and some bits of code changed accordingly...
#   - Additional regular expressions...
#   - Additional functions...
#   - Rewriting and some restructuring of the original code...
#   - Added an attempt to translate some python modules...
#     - It seems that I'm going to need a parser that builds a "rough"
#       structure of the program. I will have to keep track of all
#       variables to be able to tell what data types they hold, and
#       their scope.
#     - Problem with translation of for loops. Much due to the
#       keeping-track-of-variables problem.
#
# Here are the original notes made by Joe Strout:
#
#  python2c.py          joe@strout.net
#
#  This is a cheezy little program that converts some Python code
#  into C++ code.  It has a very limited range, but it does a good
#  job on the example code it was built around!   =)
#
#  first release: 3/21/97       JJS
#
```

```
# Changes by Dirk Heise 01-12-98            (Thanks, Dirk!)
#  - created a very simple file interface
#  - some more rules for C++ support
#  - added another "action" that may contain a python statement to
#    be executed on match, so a match can trigger something
#  - create a class header
#  - temporarily buffer output in several lists of strings
#  - added DEVELOPING option to help in examining operation of rules
#-----------------------------------------------------------------------

import re
import string

DEVELOPING = 0
  # set this to 0 for simple translation
  # set this to 1 if you want to have supplementary comments in the
  #   generated C++ code that help finding out what PYTHON2C did

# dirk heise:
# ----------------------------- RULES ------------------------------
# All these rules work on single lines of input text !
# Every rule consists of three strings (the third may be None)
# - regex
#   If you wanna create new rules:
#   the regex must contain one or more "\(.*\)" patterns,
# - replacement rule
#   the replacement rule can refer to the patterns mentioned above
#   with "^1","^2", "^3" etc. to insert the substring swallowed by that subexpr.
# - None or a Python statement to be executed on match.
#   When this statement is executed, it can refer to the subexpressions
#   eaten by the regex as "sub[1]", "sub[2]" etc.
#   you can use this to store info from the parsed text into string
#   variables.
#   IMPORTANT : the line you're defining works in a local namespace;
#   it can NOT setup global variables in this program directly
#   (i suspect it might be a bug in Python1.4)
#   My workaround suggestion:
#   When you define such a line, simply call a function you define yourself!
#   That function (see SetClassName() below) can access every global object.
# The rules are applied from top to bottom of list!
# You can exploit this by first catching special cases and later
# catch more generalized cases! (In other words, the sequence order
# might be important)
trans = [
  # Remove all invocations of the str(...) conversion method. (Tom Simonsen)
  # This will not work in regex, because there's no way of expressing that
  # parenthesis have to be balanced.
  ["(.*str\(.*\).*)" , "^1", "s = translateMethod(s, 'str', None)"],
  # COMMENTS
  # 0
  ["(.*)#(.*)", "^1//^2", None],

  # STRING LITERALS
  # 1
  ["(.*)'(.*)'(.*)", '^1"^2"^3', None],

  # WHILE LOOPS
  # 2
  ["while (.*):(.*)", "while (^1)^2", None],

  # FOR LOOPS
  # loops that iterate integers can easily be converted:
  # 3
```

```
["for (.*) in range\((.*),(.*)\):(.*)",
 "for (int ^1=^2; ^1<^3; ^1++) {^4", None],
# an attempt to make sense of loops that iterate over some sequence/list:
# (rule sequence is important here as the following rule is a superset
# of the last one!):
# 4
["for (.*) in (.*):(.*)",
 "for (int ^1i=0; ^1i<^2.Length(); ^1i++) { int ^1 = ^2[^1i]; ^3", None],
# Here, i assume that a Python sequence is represented by some
# C++ container, and this container offers a method Length()
# to find out its number of elements.
# While a Python loop does not need an int counter variable for
# this, iterating a C++ dynamic array-like container requires
# a counter int. And it requires accessing the container content
# explicitly. This rule constructs some code for that.
# Even if it doesn't compile, it'll notify you of the necessity of
# explicit indirection, it's a thing easily overlooked.
# TODO : replace Length() with something more flexible
#   or define a complete container interface somewhere...

# IF LINES
# 5
["if (.*):(.*)", "if (^1)^2", None],

# ELIF LINES (Added by Tom Simonsen)
#
["elif (.*):(.*)", "else if (^1)^2", None],

# Remove any reference to the math module. (Tom Simonsen)
#
["(.*)math\.(.*)" , "^1^2", None],

# IMPORT LINES (Added by Tom Simonsen)
#
["import (.*)", "", "s = translateImportStatements(sub[1])"],

# ELSE LINES
# 6
["else:(.*)", "else^1", None],

# PRINT LINES
# 7
["print (.*),$", "cout << ^1 << ' ';", None],
# 8
["print (.*)", "cout << ^1 << endl;", None],

# INPUT STATEMENTS
# 9
['(.*)=(.*)raw_input\("(.*)"\)(.*)',
 'cout << "^3"; cin >> ^1;^4', None],
# 10
["(.*)=(.*)raw_input\((.*)\)(.*)",
 "cin >> ^1;^4", None],
# 11
['(.*)=(.*)input\("(.*)"\)(.*)',
 'cout << "^3"; cin >> ^1;^4', None],
# 12
["(.*)=(.*)input\((.*)\)(.*)",
 "cin >> ^1;^4", None],

# C++ RULES
# some more rules by dirk heise
# MEMBER VARIABLE PREFIXES (TREATING "SELF.")
```

```
  # this is done by two rules, the sequence is important!
  # 13
  #["\(.*\)self\.\(\(\|[a-z]\|[A-Z]\)+\)(\(.*\)" , "^1^2(^4" , None],
  # this catches "self.id("
  # first catch function calls to the object itself, and simply kill
  # ".self"
  # TODO this regex fails... why? and find an easier way to catch
  #   id char set
  # 14
  ["(.*)self\.(.*)" , "^1m_^2" , None],
  # catch the rest: member variable accesses
  # this rule assumes the C++ programmer has the habit of calling member
variables
  # "m_something" (which is a habit of mine)
  # Change this rule to fit your personal C++ naming conventions!

  # CLASS DECLARATIONS
  # 15
  ["class (.*):" , "", "SetClassName(sub[1])"],
  # assign the detected class name to a global string

  # FUNCTION & METHOD DECLARATIONS
  # first catch method declarations:
  # 16
  ["def (.*)\(self\):(.*)" , "void ^c::^1()^2", None],
  # 17
  ["def (.*)\(self\),(.*)" , "void ^c::^1(^2", None],
  # put classname in front of function name, eat parameter "self"
  # the "void" is just a guess, of course.
  # TODO : ^c for classname is quite arbitrary.
  #        Setting up "classname" is okay cause its a clean way of
  #        extending but ^c is built into the translate function and
  #        it shouldn't
  #
  # now catch normal function declarations (they have no "self" argument):
  # 18
  ["def (.*)" , "void ^1", None]
  # again, the void is a guess.
  #
]
# Tom Simonsen:
# This is a part of the attempt at translating the imported python modules to
# their C++ equivalents.
modules = [
  ["[ ,]+math", "#include <math.h>\n", None]
  ]

# --------------------- EXTENSIONS -------------------------
# These variables and functions are used by user-defined python statements
# (see descriptions of rules)

header = [] # will store list of strings for class header
            # only used when a class definition is found
def hprint(s):
  # append string to class header text
  header.append(s)

classname = ""
  # dirk heise
  # global variable to keep a detected class name


def replaceGroups(match, s):
  # This piece of code was turned into a separate method by Tom Simonsen.
  # Originally a part of translate.
  pos = string.find(s,'^')
  #print "Position of ^: " + str(pos)
  while pos > -1:
    # dirk heise : special : ^c means "classname"
    # TODO: this is a nonsystematic hasty improvement hack
    #     (see the annotation in the rules section, seek TODO)
    if s[pos+1] == 'c' :
      # insert "classname" into our string
      left = s[:pos]
      right = s[pos+2:]
      k = classname
    else:
      num = string.atoi(s[pos+1:pos+2])
      #print "Number of group: " + str(num)
      # s = s[:pos] + keys[i].group(num) + s[pos+2:]
      # dirk heise : i splitted that to make it more understandable
      # for me
      left = s[:pos]
      right = s[pos+2:]
      k = match.group(num)
      if k == None :
        raise "Error in regex rule!"
    s = left + k + right

    # find another caret:
    pos = string.find(s,'^')
  return s



def translateMethod(s, method, replacement):
  # Replaces the name of a method in a method reference, with the name of
  # another method. If the replacement parameter == None, the reference
  # to the method will be deleted, along with the parenthesis.
  counter = 0
  pos = string.find(s, method)
  if pos == -1:
    return s
  number = string.count(s, method)
  for i in range(0, number, 1): # Replace all occurencies.
    if hasParenthesisToTheRight(s, pos, len(method)):
      s = changeStr(s, pos, len(method), replacement)
      s = removeParenthesisBalanced(s, pos)
      pos = string.find(s, method, pos, (len(s)-1))
  return s



def hasParenthesisToTheRight(string, position, length):
  # Checks if the sub_string in string has a "(" directly to the right
  # of it (indicating that the sub_string is in fact a part of a method
  # reference). Returns 1 if this is the case, 0 otherwise.
  c = string[position+length]
  if(c == "("):
    return 1
  else:
    return 0



def changeStr(string, position, length, replacement):
  if replacement == None:
    # Removes any substring at position with length, within string.
    return string[0:position] + string[(position+length):]
  else:
    # Replaces the existing substring with a different substring, within string.
    return string[0:position] + replacement + string[(position+length):]


def removeParenthesisBalanced(string, l_position):
  # Removes the left "(" at position and then finds the right ")",
  # keeping the parenthesis balanced. (There might be parenthesis within
  # parenthesis within parenthesis and so on.)
  p_counter = 0
  r_position = l_position
  for c in string[l_position:]:
    if c == "(":
      p_counter = p_counter + 1
    elif c == ")":
      p_counter = p_counter - 1
    if p_counter == 0:
      # The right parenthesis has been found. Now, remove the pair.
      return string[0:l_position] + string[(l_position+1):r_position] +
string[(r_position+1):]
    r_position = r_position + 1
  raise "ERROR: Unbalanced parenthesis"



def translateImportStatements(s):
  global m_keys
  global m_values
  global m_exe # TODO: Incorporate exe!
  global history
  translated_s = ""
  for i in range(0,len(m_keys)):
    match = m_keys[i].search(s)
    if match != None:
      translated_s = translated_s + m_values[i];
      history = history + "m_keys[" + str(i) + "] "
  return translated_s




def SetClassName(s):
  # dirk heise
  # set up class name , to be used in user executable statements
  # i suppose here that this function is called when a Python class
  # is defined.
  # So create some code that will work as a template for a header file
  global classname
  classname = s
  hprint ("VERY ROUGH HEADER FILE TEMPLATE FOR CLASS "+classname)
  hprint ("copy this into its own file and refine it by hand."  )
  hprint ("// "+classname+".H"                                  )
  hprint ("//"                                                  )
  hprint ("//"                                                  )
  hprint ("#ifndef _"+classname+"_H_"                           )
  hprint ("#define _"+classname+"_H_"                           )
  hprint ('#include "globs.h"'                                  )
  hprint ("class "+classname                                    )
  hprint (" {"                                                  )
  hprint ("  public:"                                           )
  hprint ("   "+classname+"();"                            )
  hprint ("   virtual ~"+classname+"();"                   )
  hprint ("  protected:"                                   )
  hprint ("  private:"                                     )
  hprint (" };"                                            )
  hprint ("#endif // _"+classname+"_H_"                    )
  hprint ("END OF HEADER FILE TEMPLATE FOR CLASS "+classname    )
    # TODO why all the mess with hprint? Well, the idea is to extend this
    #      one: First write only until destructor prototype, later
    #      when fetching a "def NAME(" "print" translation and "hprint"
    #      line as prototype (so this header file will contain
    #      more accurate info)
    #      In the end, "hprint" the rest of the header file.



test_counter = 0

# dirk heise: added parameter exe
def translate(s,keys,values,exe):
  # translate line s
  # find a match among keys
  # returns transformed "s" and a history string telling numbers of
  # transformations applied, in the form of a C++ comment
  global classname # dirk heise
  global history
  global test_counter
  changed = 1
  while changed:
    changed = 0
    for i in range(0,len(keys)):
      print str(test_counter) + " Matching string " + str(s) + " with pattern: "
+ str(keys[i].pattern)
      test_counter = test_counter + 1
      match = keys[i].match(s)
      if match != None:
        # print "*** MATCH! ***"
        # found a match ... apply translation
        history = history + "g_keys[" + str(i) + "] "
        # make sure history string entries are separated by spaces
        # to facilitate parsing these comments later (if someone wants
        # to)
        s = values[i]
        # we've got a response... stuff in adjusted text where indicated
        s = replaceGroups(match, s)
        # dirk heise : execute user statement if one is given:
        if exe[i] != None:
          # before execution, setup "environment" strings:
          sub = []
          sub.append("")
          k = " "
          num = 1
          while num != 0:
            try:
              k = match.group(num)
              num = num + 1
              sub.append(k)
            except:
              num = 0 # to quit the loop
          # sub is now a list of strings containing parsed subexpressions
          exec(exe[i])
        #changed = 1    # check for more matches!    Tom Simonsen commented this
out.
```

```
                                                   # Should be set by exe, if
desirable.

  # special case: add semicolon after most statements
  pos = string.find(s+"//", "//")
  endpos = len(string.rstrip(s[:pos])) - 1
  if s != "":
    # dirk heise: to allow rules that return an empty string
    endchar = s[endpos]
    if endpos >= 3 and s[endpos-3:endpos+1] == 'else' and (endpos == 3 or
s[endpos-4] in " \t"):
      # found dangling keyword -- no semicolon needed
      return (s," //$$$ trafos applied: "+history)
    if endpos > 0 and endchar not in "{}>;":
      s = s[:endpos+1] + ';' + s[endpos+1:]

  return (s," //$$$ trafos applied: "+history)
    # I use "//$$$" as a marker for the history string to facilitate later
    # automatic wipeaway of these comments




def processLine(s,keys,values,exe):
  # find the indentation
  global gIndents
  global translated_code
  global output
  match = re.findall("(^[\t ]*)", s)
  qtywhitechars = len(match[0])
  whitechars = match[0]

  if len(whitechars) > len(gIndents[-1]):
    indent = gIndents[-1] + "{"
    translated_code.append(indent)
    output.append(indent)
    gIndents.append(whitechars)
  else:
    while gIndents and gIndents[-1] != whitechars:
      del gIndents[-1]
      if gIndents:
          indent = gIndents[-1] + "}"
          translated_code.append(indent)
          output.append(indent)
      # if not gIndents: raise "Inconsistent indentation"
      # dirk heise: Come on... Never give up!
      if not gIndents:
        print "WARNING! Inconsistent indentation."
        gIndents.append(whitechars)

  # dirk heise: added exe , take care for history return value:
  s,history = translate(s[qtywhitechars:], keys, values, exe)
  return gIndents[-1] + s , gIndents[-1] + "  " + history




# THE MAIN CODE:

# General regex:
gKeys = map(lambda x:re.compile(x[0]), trans)
```

```
gValues = map((lambda x:x[1]), trans)
gExe = map((lambda x:x[2]), trans)
gEndWhite = re.compile("(.*)([ \t]*)$")
gIndents = ['']
# Used for module translation:
m_keys = map(lambda x:re.compile(x[0]), modules)
m_values = map((lambda x:x[1]), modules)
m_exe = map((lambda x:x[2]), modules)

s = ""

# Dirk Heise 12.01.97 : a very simple file interface.
# Modified by JJS to be platform-independent.

print "Python2C v0.01\tProgrammed by Tom Simonsen"
print "\t\t             tomsi@ifi.uio.no\n"
s = raw_input("### Write the path to the input file: ")
file_name = s
translated_code = []  # Contains the entire translated file.
output = []           # Contains the entire translated file and debugging
                      # information.
history = ""          # Temporary storage for the debugging information.
                      # history stores textual references to the changes
                      # that have been made by what regular expressions.
                      # Set DEVELOPING = 1 to get the output.

try:
  f = open(s)
  print "### Translating..."
  lines = f.readlines()
  for s in lines:
    translated_line, history = processLine(s, gKeys, gValues,gExe)
    translated_code.append(translated_line)
    output.append(translated_line)
    if DEVELOPING:
      # print numbers of transformations applied
      output.append(history)
    history = ""

  # now output class header if there is one:
  for s in header:
    print s

  print "### COMPLETED translation of the file \"" + file_name + "\".\n"
  print "### Beginning of file."
  for line in output:
    print line
  print "### End of file.\n"

except IOError:
  print "### File \"" + file_name + "\" not found!"
```

6.2 A scanner and parser for the Pedagola language

This chapter includes all the source code that went into making my scanner and parser for the Pedagola language. I have only included the source code that was written by hand and not the Lex and Yacc output files. The output files from Lex and Yacc go into the actual source code of the scanner/parser program itself, the input files do not. This is not a complete implementation of a Pedagola language scanner and parser, nor is it meant to be. It only serves as an example of the amount of work that goes into making a scanner and parser from scratch using the Lex and Yacc generators. Pedagola is a very small and limited language. Its grammar does not come close to the complex grammar of a full-blown, real-world programming language.

This is the contents of the Lex input file for making the scanner for the Pedagola language. The grammar for the Pedagola language can be found at the University of Oslo's website, http://www.ifi.uio.no/inf310/oblig-sprak-2004.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "y.tab.h"
#include "datastructure.h"

const int DEBUG_L = 0;
size_t CONST_LENGTH = 16;
extern int current_type;
extern char* current_name;
extern struct table_entry* current_proc;

int counter = 0, line_counter = 1;
int last_scanned_token;

%}
non_zero_digit [1-9]
digits [0-9]
number ({non_zero_digit}{digits}*("."{digits}+)?)|("0."{digits}+)|"0"
true "true"
false "false"
const {number}|{true}|{false}
alpha [a-zA-Z]
name {alpha}+[0-9a-zA-Z]*
whitespace [ \t]+
newline "\n"
semi ";"
assign ":="
less "<"
lessequal "<="
larger ">"
largerequal ">="
equal "="
notequal "=/="
orop "or"
andop "and"
notop "not"
addop "+"
subop "-"
mulop "*"
divop "/"
powop "**"
prog "prog"
proc "proc"
begprog "begprog"
endprog "endprog"
begproc "begproc"
endproc "endproc"
if "if"
then "then"
else "else"
endif "fi"
```

```
real "real"
bool "bool"
void "void"
while "while"
do "do"
enddo "od"
return "return"
comment "--"[^\n]*
leftp "("
rightp ")"
comma ","
%%
{whitespace} {
  if(DEBUG_L)
    printf("Line %d: whitespace = \"%s\"\n", line_counter, yytext);
}
{newline} {
  if(DEBUG_L)
    printf("Line %d: newline = \"%s\"\n", line_counter, yytext);
  line_counter++;
}
{comment} {
  if(DEBUG_L)
    printf("Line %d: comment = \"%s\"\n", line_counter, yytext);
}
{true} {
  struct table_entry *entry;
  entry=(struct table_entry*)malloc(sizeof(struct table_entry));
  if(DEBUG_L)
    printf("Line %d: true = \"%s\"\n", line_counter, yytext);
  entry->real_value = 0;
  entry->bool_value = 1;
  entry->type = BOOL;
  entry->is_func = 0;
  entry->has_param = 0;
  entry->param_type = 0;
  entry->left = NULL;
  entry->right = NULL;
  yylval.te = entry;
  last_scanned_token = CONST;
  return CONST;
}
{false} {
  struct table_entry *entry;
  entry=(struct table_entry*)malloc(sizeof(struct table_entry));
  if(DEBUG_L)
    printf("Line %d: false = \"%s\"\n", line_counter, yytext);
  entry->real_value = 0;
  entry->bool_value = 0;
  entry->type = BOOL;
  entry->is_func = 0;
  entry->has_param = 0;
  entry->param_type = 0;
  entry->left = NULL;
  entry->right = NULL;
  yylval.te = entry;
  last_scanned_token = CONST;
  return CONST;
}
{semi} {
  if(DEBUG_L)
    printf("Line %d: semi = \"%s\"\n", line_counter, yytext);
  last_scanned_token = SEMI;
  return SEMI;
}
{assign} {
  struct table_entry *entry;
  entry=(struct table_entry*)malloc(sizeof(struct table_entry));
  if(DEBUG_L)
    printf("Line %d: assign = \"%s\"\n", line_counter, yytext);
  char* n = malloc(sizeof(yytext));
  memset(n, '\0', strlen(yytext));
  const char* copy = yytext;
  strcpy(n, copy);
  entry->name = n;
  entry->real_value = 0;
  entry->bool_value = 0;
  entry->type = ASSIGN;
  entry->is_func = 0;
  entry->has_param = 0;
  entry->param_type = 0;
  entry->left = NULL;
  entry->right = NULL;
  yylval.te = entry;
  last_scanned_token = ASSIGN;
  return ASSIGN;
}
{less} {
  struct table_entry *entry;
  entry=(struct table_entry*)malloc(sizeof(struct table_entry));
  if(DEBUG_L)
    printf("Line %d: less = \"%s\"\n", line_counter, yytext);
  char* n = malloc(sizeof(yytext));
  memset(n, '\0', strlen(yytext));
  const char* copy = yytext;
  strcpy(n, copy);
  entry->name = n;
  entry->real_value = 0;
  entry->bool_value = 0;
  entry->type = LESS;
  entry->is_func = 0;
  entry->has_param = 0;
  entry->param_type = 0;
  entry->left = NULL;
  entry->right = NULL;
  yylval.te = entry;
  last_scanned_token = LESS;
  return LESS;
}
{lessequal} {
  struct table_entry *entry;
  entry=(struct table_entry*)malloc(sizeof(struct table_entry));
  if(DEBUG_L)
    printf("Line %d: lessequal = \"%s\"\n", line_counter, yytext);
  char* n = malloc(sizeof(yytext));
  memset(n, '\0', strlen(yytext));
  const char* copy = yytext;
  strcpy(n, copy);
  entry->name = n;
  entry->real_value = 0;
  entry->bool_value = 0;
  entry->type = LESSEQUAL;
  entry->is_func = 0;
  entry->has_param = 0;
  entry->param_type = 0;
  entry->left = NULL;
  entry->right = NULL;
  yylval.te = entry;
  last_scanned_token = LESSEQUAL;
  return LESSEQUAL;
}
{larger} {
  struct table_entry *entry;
  entry=(struct table_entry*)malloc(sizeof(struct table_entry));
  if(DEBUG_L)
    printf("Line %d: larger = \"%s\"\n", line_counter, yytext);
  char* n = malloc(sizeof(yytext));
  memset(n, '\0', strlen(yytext));
  const char* copy = yytext;
  strcpy(n, copy);
  entry->name = n;
  entry->real_value = 0;
  entry->bool_value = 0;
  entry->type = LARGER;
  entry->is_func = 0;
  entry->has_param = 0;
  entry->param_type = 0;
  entry->left = NULL;
  entry->right = NULL;
  yylval.te = entry;
  last_scanned_token = LARGER;
  return LARGER;
}
{largerequal} {
  struct table_entry *entry;
  entry=(struct table_entry*)malloc(sizeof(struct table_entry));
  if(DEBUG_L)
    printf("Line %d: largerequal = \"%s\"\n", line_counter, yytext);
  char* n = malloc(sizeof(yytext));
  memset(n, '\0', strlen(yytext));
  const char* copy = yytext;
  strcpy(n, copy);
  entry->name = n;
  entry->real_value = 0;
  entry->bool_value = 0;
  entry->type = LARGEREQUAL;
  entry->is_func = 0;
  entry->has_param = 0;
  entry->param_type = 0;
  entry->left = NULL;
  entry->right = NULL;
  yylval.te = entry;
  last_scanned_token = LARGEREQUAL;
  return LARGEREQUAL;
}
{equal} {
  struct table_entry *entry;
  entry=(struct table_entry*)malloc(sizeof(struct table_entry));
  if(DEBUG_L)
    printf("Line %d: equal = \"%s\"\n", line_counter, yytext);
  char* n = malloc(sizeof(yytext));
  memset(n, '\0', strlen(yytext));
  const char* copy = yytext;
  strcpy(n, copy);
  entry->name = n;
  entry->real_value = 0;
  entry->bool_value = 0;
  entry->type = EQUAL;
  entry->is_func = 0;
  entry->has_param = 0;
  entry->param_type = 0;
  entry->left = NULL;
  entry->right = NULL;
  yylval.te = entry;
  last_scanned_token = EQUAL;
  return EQUAL;
}
{notequal} {
  struct table_entry *entry;
  entry=(struct table_entry*)malloc(sizeof(struct table_entry));
  if(DEBUG_L)
    printf("Line %d: notequal = \"%s\"\n", line_counter, yytext);
  char* n = malloc(sizeof(yytext));
  memset(n, '\0', strlen(yytext));
  const char* copy = yytext;
  strcpy(n, copy);
  entry->name = n;
  entry->real_value = 0;
  entry->bool_value = 0;
  entry->type = NOTEQUAL;
  entry->is_func = 0;
  entry->has_param = 0;
  entry->param_type = 0;
  entry->left = NULL;
  entry->right = NULL;
  yylval.te = entry;
  last_scanned_token = NOTEQUAL;
  return NOTEQUAL;
}
{orop} {
  struct table_entry *entry;
  entry=(struct table_entry*)malloc(sizeof(struct table_entry));
  if(DEBUG_L)
    printf("Line %d: orop = \"%s\"\n", line_counter, yytext);
  char* n = malloc(sizeof(yytext));
  memset(n, '\0', strlen(yytext));
  const char* copy = yytext;
  strcpy(n, copy);
  entry->name = n;
  entry->real_value = 0;
  entry->bool_value = 0;
  entry->type = OROP;
  entry->is_func = 0;
  entry->has_param = 0;
  entry->param_type = 0;
  entry->left = NULL;
  entry->right = NULL;
  yylval.te = entry;
  last_scanned_token = OROP;
  return OROP;
}
{notop} {
  struct table_entry *entry;
  entry=(struct table_entry*)malloc(sizeof(struct table_entry));
  if(DEBUG_L)
    printf("Line %d: notop = \"%s\"\n", line_counter, yytext);
  char* n = malloc(sizeof(yytext));
  memset(n, '\0', strlen(yytext));
  const char* copy = yytext;
  strcpy(n, copy);
  entry->name = n;
  entry->real_value = 0;
  entry->bool_value = 0;
  entry->type = NOTOP;
  entry->is_func = 0;
  entry->has_param = 0;
  entry->param_type = 0;
  entry->left = NULL;
  entry->right = NULL;
  yylval.te = entry;
  last_scanned_token = NOTOP;
  return NOTOP;
}
{addop} {
  struct table_entry *entry;
  entry=(struct table_entry*)malloc(sizeof(struct table_entry));
  if(DEBUG_L)
    printf("Line %d: addop = \"%s\"\n", line_counter, yytext);
  char* n = malloc(sizeof(yytext));
  memset(n, '\0', strlen(yytext));
  const char* copy = yytext;
  strcpy(n, copy);
  entry->name = n;
  entry->real_value = 0;
  entry->bool_value = 0;
  entry->type = ADDOP;
  entry->is_func = 0;
  entry->has_param = 0;
  entry->param_type = 0;
  entry->left = NULL;
  entry->right = NULL;
  yylval.te = entry;
  last_scanned_token = ADDOP;
  return ADDOP;
}
{subop} {
  struct table_entry *entry;
  entry=(struct table_entry*)malloc(sizeof(struct table_entry));
  if(DEBUG_L)
    printf("Line %d: subop = \"%s\"\n", line_counter, yytext);
  char* n = malloc(sizeof(yytext));
  memset(n, '\0', strlen(yytext));
  const char* copy = yytext;
  strcpy(n, copy);
  entry->name = n;
  entry->real_value = 0;
  entry->bool_value = 0;
  entry->type = SUBOP;
  entry->is_func = 0;
  entry->has_param = 0;
  entry->param_type = 0;
  entry->left = NULL;
  entry->right = NULL;
  yylval.te = entry;
  last_scanned_token = SUBOP;
  return SUBOP;
}
{andop} {
  struct table_entry *entry;
  entry=(struct table_entry*)malloc(sizeof(struct table_entry));
  if(DEBUG_L)
    printf("Line %d: andop = \"%s\"\n", line_counter, yytext);
  char* n = malloc(sizeof(yytext));
  memset(n, '\0', strlen(yytext));
  const char* copy = yytext;
  strcpy(n, copy);
  entry->name = n;
  entry->real_value = 0;
  entry->bool_value = 0;
  entry->type = ANDOP;
  entry->is_func = 0;
  entry->has_param = 0;
  entry->param_type = 0;
  entry->left = NULL;
  entry->right = NULL;
  yylval.te = entry;
  last_scanned_token = ANDOP;
  return ANDOP;
}
{mulop} {
  struct table_entry *entry;
  entry=(struct table_entry*)malloc(sizeof(struct table_entry));
  if(DEBUG_L)
    printf("Line %d: mulop = \"%s\"\n", line_counter, yytext);
  char* n = malloc(sizeof(yytext));
  memset(n, '\0', strlen(yytext));
  const char* copy = yytext;
  strcpy(n, copy);
  entry->name = n;
  entry->real_value = 0;
  entry->bool_value = 0;
  entry->type = MULOP;
  entry->is_func = 0;
  entry->has_param = 0;
  entry->param_type = 0;
  entry->left = NULL;
  entry->right = NULL;
  yylval.te = entry;
  last_scanned_token = MULOP;
  return MULOP;
}
{divop} {
  struct table_entry *entry;
  entry=(struct table_entry*)malloc(sizeof(struct table_entry));
  if(DEBUG_L)
    printf("Line %d: divop = \"%s\"\n", line_counter, yytext);
  char* n = malloc(sizeof(yytext));
  memset(n, '\0', strlen(yytext));
  const char* copy = yytext;
  strcpy(n, copy);
  entry->name = n;
  entry->real_value = 0;
  entry->bool_value = 0;
  entry->type = DIVOP;
  entry->is_func = 0;
  entry->has_param = 0;
  entry->param_type = 0;
  entry->left = NULL;
  entry->right = NULL;
  yylval.te = entry;
  last_scanned_token = DIVOP;
  return DIVOP;
}
{powop} {
  struct table_entry *entry;
  entry=(struct table_entry*)malloc(sizeof(struct table_entry));
  if(DEBUG_L)
    printf("Line %d: powop = \"%s\"\n", line_counter, yytext);
  char* n = malloc(sizeof(yytext));
  memset(n, '\0', strlen(yytext));
  const char* copy = yytext;
  strcpy(n, copy);
  entry->name = n;
  entry->real_value = 0;
  entry->bool_value = 0;
  entry->type = POWOP;
  entry->is_func = 0;
  entry->has_param = 0;
  entry->param_type = 0;
  entry->left = NULL;
  entry->right = NULL;
  yylval.te = entry;
  last_scanned_token = POWOP;
  return POWOP;
}
{prog} {
  if(DEBUG_L)
    printf("Line %d: prog = \"%s\"\n", line_counter, yytext);
  push(); /* Creates the bottom stack_entry
           in the symbol stack.*/
  last_scanned_token = PROG;
  return PROG;
}
{proc} {
  if(DEBUG_L)
    printf("Line %d: proc = \"%s\"\n", line_counter, yytext);
  last_scanned_token = PROC;
  return PROC;
}
{begprog} {
  if(DEBUG_L)
    printf("Line %d: begprog = \"%s\"\n", line_counter, yytext);
  last_scanned_token = BEGPROG;
  return BEGPROG;
}
{endprog} {
  if(DEBUG_L)
    printf("Line %d: endprog = \"%s\"\n", line_counter, yytext);
  last_scanned_token = ENDPROG;
  return ENDPROG;
}
{begproc} {
  if(DEBUG_L)
    printf("Line %d: begproc = \"%s\"\n", line_counter, yytext);
  last_scanned_token = BEGPROC;
  return BEGPROC;
}
{endproc} {
  if(DEBUG_L)
    printf("Line %d: endproc = \"%s\"\n", line_counter, yytext);
  last_scanned_token = ENDPROC;
  return ENDPROC;
}
{if} {
  if(DEBUG_L)
    printf("Line %d: if = \"%s\"\n", line_counter, yytext);
  last_scanned_token = IF;
  return IF;
}
{then} {
  if(DEBUG_L)
```

```
    printf("Line %d: then = \"%s\"\n", line_counter, yytext);
  last_scanned_token = THEN;
  return THEN;
}
{else} {
  if(DEBUG_L)
    printf("Line %d: else = \"%s\"\n", line_counter, yytext);
  last_scanned_token = ELSE;
  return ELSE;
}
{endif} {
  if(DEBUG_L)
    printf("Line %d: endif = \"%s\"\n", line_counter, yytext);
  last_scanned_token = ENDIF;
  return ENDIF;
}
{real} {
  if(DEBUG_L)
    printf("Line %d: real = \"%s\"\n", line_counter, yytext);
  yylval.type = REAL;
  last_scanned_token = REAL;
  current_type = REAL;
  return REAL;
}
{bool} {
  if(DEBUG_L)
    printf("Line %d: bool = \"%s\"\n", line_counter, yytext);
  yylval.type = BOOL;
  last_scanned_token = BOOL;
  current_type = BOOL;
  return BOOL;
}
{void} {
  if(DEBUG_L)
    printf("Line %d: void = \"%s\"\n", line_counter, yytext);
  yylval.type = VOID;
  last_scanned_token = VOID;
  current_type = VOID;
  return VOID;
}
{while} {
  if(DEBUG_L)
    printf("Line %d: while = \"%s\"\n", line_counter, yytext);
  last_scanned_token = WHILE;
  return WHILE;
}
{do} {
  if(DEBUG_L)
    printf("Line %d: do = \"%s\"\n", line_counter, yytext);
  last_scanned_token = DO;
  return DO;
}
{enddo} {
  if(DEBUG_L)
    printf("Line %d: enddo = \"%s\"\n", line_counter, yytext);
  last_scanned_token = ENDDO;
  return ENDDO;
}
{return} {
  if(DEBUG_L)
    printf("Line %d: return = \"%s\"\n", line_counter, yytext);
  last_scanned_token = RETURN;
  return RETURN;
```

```
}
{number} {
  struct table_entry* entry;
  entry=(struct table_entry*)malloc(sizeof(struct table_entry));
  if(DEBUG_L)
    printf("Line %d: number = \"%s\"\n", line_counter, yytext);
  if(strlen(yytext) > CONST_LENGTH)
    {
      printf("Scanning error:\nNumber cannot exceed 16 characters.(\"%s\")\n",
yytext);
      exit(1);
    }
  else
    {
      entry->name = NULL;
      entry->real_value = atoi(yytext);
      entry->bool_value = 0;
      entry->type = REAL;
      entry->is_func = 0;
      entry->has_param = 0;
      entry->param_type = 0;
      entry->left = NULL;
      entry->right = NULL;
      yylval.te = entry;
      last_scanned_token = CONST;
      return CONST;
    }
}
{name} {
  if(DEBUG_L)
    printf("Line %d: name = \"%s\"\n", line_counter, yytext);
  if(strlen(yytext) > CONST_LENGTH)
    {
      printf("Scanning error:\nName cannot exceed 16 characters.(\"%s\")\n",
yytext);
      exit(1);
    }
  else
    {
      struct table_entry* entry;
      entry = (struct table_entry*)malloc(sizeof(struct table_entry));
      entry->bool_value = 0;
      entry->real_value = 0;
      entry->type = NAME;
      entry->is_func = 0;
      entry->has_param = 0;
      entry->param_type = 0;
      entry->left = NULL;
      entry->right = NULL;
      if(last_scanned_token == PROC)
      entry->is_func = 1;

      /* Copy the yytext string to a separate memory location: */
      char* n = malloc(sizeof(yytext));
      memset(n, '\0', strlen(yytext));
      const char* copy = yytext;
      strcpy(n, copy);

      entry->name =  n;
      current_name = n;
      if(last_scanned_token == REAL || last_scanned_token == BOOL ||
       last_scanned_token == VOID || last_scanned_token == PROC ||
       last_scanned_token == COMMA)
      {
        /* We are currently scanning a variable or function declaration. */
        entry->type = current_type;
        if(existsInTop(yytext))
          {
            printf("Error on line %d: Variable or function name already in use
(Name = \"%s\").\n", line_counter, yytext);
            exit(1);
          }
        else
          addName(entry, current_type);
      }
      else
      {
        /* We are currently scanning a variable or function reference. */
        if(!existsInStack(yytext))
          {
            printf("Error on line %d: Referencing undeclared variable or
function (Name = \"%s\").\n", line_counter, yytext);
            exit(1);
          }
      }
      yylval.te = entry;
      if(last_scanned_token == PROC)
      {
        current_proc = entry;
        push();
      }
      if(DEBUG_L)
      printStack();
      last_scanned_token = NAME;
      return NAME;
    }
}
{leftp} {
  if(DEBUG_L)
    printf("Line %d: leftp = \"%s\"\n", line_counter, yytext);
  last_scanned_token = LEFTP;
  return LEFTP;
}
{rightp} {
  if(DEBUG_L)
    printf("Line %d: rightp = \"%s\"\n", line_counter, yytext);
  last_scanned_token = RIGHTP;
  return RIGHTP;
}
{comma} {
  if(DEBUG_L)
    printf("Line %d: comma = \"%s\"\n", line_counter, yytext);
  last_scanned_token = COMMA;
  return COMMA;
}
.  {
  printf("Scanner error on line %d: Unrecognized token \"%s\".\n", line_counter,
yytext);
  exit(1);
}
%%
```

This is the contents of the Yacc input. Yacc uses this file to generate the parser for the Pedagola language.

```
#include "lex.yy.c"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "datastructure.h"

const int DEBUG_Y = 0;
int current_type = 0;
char* current_name = "";
struct table_entry* current_proc;
extern int line_counter;

%}
%union {
  struct table_entry* te;
  int type;
}
%token <te> NAME
%token <te> CONST
%token SEMI
%token <te> ASSIGN
%token <te> LESS
%token <te> LESSEQUAL
%token <te> LARGER
%token <te> LARGEREQUAL
%token <te> EQUAL
%token <te> NOTEQUAL
%token <te> OROP
%token <te> NOTOP
%token <te> ANDOP
%token <te> ADDOP
%token <te> SUBOP
%token <te> MULOP
%token <te> DIVOP
%token <te> POWOP
%token PROG
%token PROC
%token BEGPROG
%token ENDPROG
%token BEGPROC
%token ENDPROC
%token IF
%token THEN
%token ELSE
%token ENDIF
%token <type> REAL
%token <type> BOOL
%token <type> VOID
%token WHILE
%token DO
%token ENDDO
%token RETURN
%token LEFTP
%token RIGHTP
%token COMMA
%type  <te> STMT
%type  <te> EXPM
%type  <te> EXP
%type  <te> PARAM
%type  <te> EXPPART
%type  <te> CALL
```

```
%type  <te> TERM1
%type  <te> TERM2
%type  <te> TERM3
%type  <te> TERM4
%type  <te> TERM5
%type  <te> TERM6
%type  <te> FACTOR
%type  <te> MULDIV
%type  <te> ADDSUB
%type  <te> RELOP
%%
PROGRAM : PROG DECL BEGPROG STMTSM ENDPROG
          {
          if(DEBUG_Y)
            printf("### Parser: program\n");
          pop(); /* The last pop on the stack.
                    The stack is now empty. */
        }
        ;
DECL : /* nothing */
       {
       if(DEBUG_Y)
         printf("### Parser: decl1\n");
       }
     | VARS
       {
       if(DEBUG_Y)
         printf("### Parser: decl2\n");
       }
     ;
VAR : REAL NAMES SEMI
      {
      if(DEBUG_Y)
        printf("### Parser: var1\n");
      }
    | BOOL NAMES SEMI
      {
      if(DEBUG_Y)
        printf("### Parser: var2\n");
      }
    | REAL PROC NAME PARAM DECL BEGPROC STMTSM ENDPROC SEMI
      {
      struct table_entry* entry;
      if(DEBUG_Y)
        printf("### Parser: decl2\n");
      entry = getTableEntry($3->name);
      if(entry != NULL && $4 != NULL)
        {
          entry->has_param = 1;
          entry->param_type = $4->type;
        }
      current_proc = NULL;
      pop();
      }
    | BOOL PROC NAME PARAM DECL BEGPROC STMTSM ENDPROC SEMI
      {
      struct table_entry* entry;
      if(DEBUG_Y)
        printf("### Parser: decl3\n");
      entry = getTableEntry($3->name);
      if(entry != NULL && $4 != NULL)
        {
          entry->has_param = 1;
```

```
            entry->param_type = $4->type;
          }
        current_proc = NULL;
        pop();
        }
      | VOID PROC NAME PARAM DECL BEGPROC STMTSM ENDPROC SEMI
        {
        struct table_entry* entry;
        if(DEBUG_Y)
          printf("### Parser: decl4\n");
        entry = getTableEntry($3->name);
        if(entry != NULL && $4 != NULL)
          {
            entry->has_param = 1;
            entry->param_type = $4->type;
          }
        current_proc = NULL;
        pop();
        }
      ;
VARS : VAR
         {
         if(DEBUG_Y)
           printf("### Parser: vars1\n");
         }
       | VARS VAR
         {
         if(DEBUG_Y)
           printf("### Parser: vars2\n");
         }
       ;
NAMES : NAME
          {
          if(DEBUG_Y)
            printf("### Parser: names1\n");
          if($1->is_func)
           {
             printf("Error on line %d: Function \"%s\" used as a variable
reference.\n", line_counter, $1->name);
             exit(1);
           }
        }

        | NAMES COMMA NAME
          {
          if(DEBUG_Y)
            printf("### Parser: names2\n");
        }
        ;
PARAM : LEFTP RIGHTP
          {
          if(DEBUG_Y)
            printf("### Parser: param1\n");
          $$ = NULL;
        }
        | LEFTP REAL NAME RIGHTP
          {
          struct table_entry* entry;
          if(DEBUG_Y)
            printf("### Parser: param2\n");
          entry = getTableEntry($3->name);
          if(entry != NULL)
            $$ = $3;
        }
        | LEFTP BOOL NAME RIGHTP
          {
          struct table_entry* entry;
          if(DEBUG_Y)
            printf("### Parser: param3\n");
          entry = getTableEntry($3->name);
          if(entry != NULL)
            $$ = $3;
        }
        ;
  STMT : NAME ASSIGN EXPM
         {
         struct table_entry* entry;
         if(DEBUG_Y)
           printf("### Parser: stmt1\n");
         if($1->is_func)
           {
             printf("Error on line %d: Function \"%s\" used as a variable
  reference.\n", line_counter, $1->name);
             exit(1);
           }
         entry = getTableEntry($1->name);
         if(entry != NULL)
           {
             if(entry->type != $3->type)
               {
               printf("Error on line %d: Wrong type in assignment.\n",
  line_counter);
               exit(1);
               }
           }
         }
       | IF EXPM THEN STMTSM ELSESTMT ENDIF
         {
         if(DEBUG_Y)
           printf("### Parser: stmt2\n");
         if($2->type != BOOL)
           {
             printf("Error on line %d: Expression after \"if\" must return
  bool.\n", line_counter);
             exit(1);
           }
         }
       | WHILE EXPM DO STMTSM ENDDO
         {
         if(DEBUG_Y)
           printf("### Parser: stmt3\n");
         if($2->type != BOOL)
           {
             printf("Error on line %d: Expression after \"while\" must return
  bool.\n", line_counter);
             exit(1);
           }
         }
       | RETURN EXPPART
         {
         if(DEBUG_Y)
           printf("### Parser: stmt4\n");
         if(current_proc == NULL)
           {
             printf("Error on line %d: Cannot have return statement outside
  function.\n", line_counter);
```

```
            exit(1);
          }
        if($2 == NULL)
          {
            if(current_proc->type != VOID)
              {
              printf("Error on line %d: Void function \"%s\" cannot return a
value.\n", line_counter, current_proc->name);
              exit(1);
              }
          }
        else
          {
            if(current_proc->type == VOID)
              {
              printf("Error on line %d: Function \"%s\" not declared as void.\n",
line_counter, current_proc->name);
              exit(1);
              }
          }
        }
      | CALL
        {
        struct table_entry* entry;
        if(DEBUG_Y)
          printf("### Parser: stmt5\n");
        entry = getTableEntry($1->name);
        if(entry != NULL)
          {
            if(entry->type != VOID)
              {
              printf("Error on line %d: Illegal use of function call. ",
line_counter);
              if(entry->type == REAL)
                printf("Function \"%s\" declared as real.\n", $1->name);
              else if(entry->type == BOOL)
                printf("Function \"%s\" declared as bool.\n", $1->name);
              exit(1);
              }
          }
        }
      ;
STMTSM : /* nothing */
         {
         if(DEBUG_Y)
           printf("### Parser: stmtsm1\n");
       }
        | STMTS
         {
         if(DEBUG_Y)
           printf("### Parser: stmtsm2\n");
       }
        ;
STMTS : STMT SEMI
         {
         if(DEBUG_Y)
           printf("### Parser: stmts1\n");
       }
       | STMTS STMT SEMI
         {
         if(DEBUG_Y)
           printf("### Parser: stmts2\n");
       }
      ;
ELSESTMT : /* nothing */
            {
            if(DEBUG_Y)
              printf("### Parser: elsestmt1\n");
          }
          | ELSE STMTSM
            {
            if(DEBUG_Y)
              printf("### Parser: elsestmt2\n");
          }
          ;
EXPPART : /* nothing */
           {
           if(DEBUG_Y)
             printf("### Parser: exppart1\n");
           $$ = NULL;
         }
         | EXPM
           {
           if(DEBUG_Y)
             printf("### Parser: exppart2\n");
           $$ = $1;
         }
         ;
ADDSUB : ADDOP
         {
         if(DEBUG_Y)
           printf("### Parser: addsub1\n");
      }
       | SUBOP
         {
         if(DEBUG_Y)
           printf("### Parser: addsub2\n");
       }
       ;
MULDIV : MULOP
         {
         if(DEBUG_Y)
           printf("### Parser: muldiv1\n");
       }
       | DIVOP
         {
         if(DEBUG_Y)
           printf("### Parser: muldiv2\n");
       }
       ;
RELOP : LESS
        {
        if(DEBUG_Y)
          printf("### Parser: less\n");
      }
      | LESSEQUAL
        {
        if(DEBUG_Y)
          printf("### Parser: lessequal1\n");
      }
      | LARGER
        {
        if(DEBUG_Y)
          printf("### Parser: larger\n");
      }
      | LARGEREQUAL
```

```
        {
        if(DEBUG_Y)
          printf("### Parser: largerequal\n");
      }
      | EQUAL
        {
        if(DEBUG_Y)
          printf("### Parser: equal\n");
      }
      | NOTEQUAL
        {
        if(DEBUG_Y)
          printf("### Parser: notequal\n");
      }
      ;
EXPM : EXP
       {
       if(DEBUG_Y)
          printf("### Parser: expm\n");
       $$ = $1;
       printf("\n### Expression found:\n");
       printTree($1);
       printf("\n### End of expression\n\n");
       }
     ;
EXP : EXP OROP TERM1
      {
      if(DEBUG_Y)
        printf("### Parser: exp1\n");
      if($1->type != BOOL)
        {
          printf("Error on line %d: \"%s\" has wrong type.\n", line_counter, $1-
>name);
          exit(1);
        }
      if($3->type != BOOL)
        {
          printf("Error on line %d: \"%s\" has wrong type.\n", line_counter, $3-
>name);
          exit(1);
        }
        $2->left = $1;
      $2->right = $3;
      $2->type = BOOL;
      $$ = $2;
      }
    | TERM1
      {
      if(DEBUG_Y)
        printf("### Parser: exp2\n");
      $$ = $1;
      }
    ;
TERM1 : TERM1 ANDOP TERM2
        {
        if(DEBUG_Y)
          printf("### Parser: term11\n");
        if($1->type != BOOL)
          {
            printf("Error on line %d: Left \"%s\" has wrong type.\n",
line_counter, $1->name);
            exit(1);
          }
        if($3->type != BOOL)
          {
            printf("Error on line %d: Right \"%s\" has wrong type.\n",
line_counter, $3->name);
            exit(1);
          }
        $2->left = $1;
        $2->right = $3;
        $2->type = BOOL;
        $$ = $2;
      }
      | TERM2
        {
        if(DEBUG_Y)
          printf("### Parser: term12\n");
        $$ = $1;
      }
      ;
TERM2 : NOTOP TERM3
        {
        if(DEBUG_Y)
          printf("### Parser: term21\n");
        if($2->type != BOOL)
          {
            printf("Error on line %d: \"%s\" has wrong type.\n", line_counter,
$2->name);
            exit(1);
          }
        $1->right = $2;
        $1->type = BOOL;
        $$ = $1;
      }
      | TERM3
        {
        if(DEBUG_Y)
          printf("### Parser: term22\n");
        $$ = $1;
      }
      ;
TERM3 : TERM4 RELOP TERM4
        {
        if(DEBUG_Y)
          printf("### Parser: term31\n");
        if($1->type != REAL)
          {
            printf("Error on line %d: \"%s\" has wrong type.\n", line_counter,
$1->name);
            exit(1);
          }
        if($3->type != REAL)
          {
            printf("Error on line %d: \"%s\" has wrong type.\n", line_counter,
$3->name);
            exit(1);
          }
        $2->left = $1;
        $2->right = $3;
        $2->type = BOOL;
        $$ = $2;
      }
      | TERM4
        {
        if(DEBUG_Y)
```

```
          printf("### Parser: term32\n");
        $$ = $1;
      }
      ;
TERM4 : TERM4 ADDSUB TERM5
        {
        if(DEBUG_Y)
          printf("### Parser: term41\n");
        if($1->type != REAL)
          {
            printf("Error on line %d: \"%s\" has wrong type.\n", line_counter,
$1->name);
            exit(1);
          }
        if($3->type != REAL)
          {
            printf("Error on line %d: \"%s\" has wrong type.\n", line_counter,
$3->name);
            exit(1);
          }
        $2->left = $1;
        $2->right = $3;
        $2->type = REAL;
        $$ = $2;
      }
      | TERM5
        {
        if(DEBUG_Y)
          printf("### Parser: term42\n");
        $$ = $1;
      }
      ;
TERM5 : TERM5 MULDIV TERM6
        {
        if(DEBUG_Y)
          printf("### Parser: term51\n");
        if($1->type != REAL)
          {
            printf("Error on line %d: \"%s\" has wrong type.\n", line_counter,
$1->name);
            exit(1);
          }
        if($3->type != REAL)
          {
            printf("Error on line %d: \"%s\" has wrong type.\n", line_counter,
$3->name);
            exit(1);
          }
        $2->left = $1;
        $2->right = $3;
        $2->type = REAL;
        $$ = $2;
      }
      | TERM6
        {
        if(DEBUG_Y)
          printf("### Parser: term52\n");
        $$ = $1;
      }
      ;
TERM6 : FACTOR POWOP TERM6
        {
        if(DEBUG_Y)
```

```
          printf("### Parser: term61\n");
        if($1->type != REAL)
          {
            printf("Error on line %d: \"%s\" has wrong type.\n", line_counter,
$1->name);
            exit(1);
          }
        if($3->type != REAL)
          {
            printf("Error on line %d: \"%s\" has wrong type.\n", line_counter,
$3->name);
            exit(1);
          }
        $2->left = $1;
        $2->right = $3;
        $2->type = REAL;
        $$ = $2;
      }
      | FACTOR
        {
        if(DEBUG_Y)
          printf("### Parser: term62\n");
        $$ = $1;
      }
      ;
FACTOR : LEFTP EXP RIGHTP
         {
         if(DEBUG_Y)
           printf("### Parser: factor1\n");
         $$ = $2;
       }
       | NAME
         {
         struct table_entry* entry;
         if(DEBUG_Y)
           printf("### Parser: factor2\n");
         entry = getTableEntry($1->name);
         if(entry != NULL)
           $$ = entry;
       }
       | CONST
         {
         if(DEBUG_Y)
           printf("### Parser: factor3\n");
         $$ = $1;
       }
       | CALL
         {
         struct table_entry* entry;
         if(DEBUG_Y)
           printf("### Parser: factor4\n");
         entry = getTableEntry($1->name);
         if(entry != NULL)
           {
             if(entry->type == VOID)
             {
               printf("Error on line %d: Illegal use of function call. ",
line_counter);
               printf("Function \"%s\" declared as void.\n", $1->name);
               exit(1);
             }
             $$ = $1;
           }
       }
       ;
CALL : NAME LEFTP EXPPART RIGHTP
       {
       struct table_entry* entry;
       if(DEBUG_Y)
         printf("### Parser: call\n");
       entry = getTableEntry($1->name);
       if(entry != NULL)
         {
           if(!(entry->is_func))
             {
             printf("Error on line %d: Variable \"%s\" referenced as a function
call.\n", line_counter, entry->name);
             exit(1);
             }
           if($3 == NULL)
             {
             if(entry->has_param)
               {
                 printf("Error on line %d: Function \"%s\" called without
parameter.\n", line_counter, entry->name);
                 exit(1);
               }
             }
           else
             {
             if(!(entry->has_param))
               {
                 printf("Error on line %d: Function \"%s\" has no parameter.\n",
line_counter, entry->name);
                 exit(1);
               }
             else
               {
                 if(entry->param_type != $3->type)
                   {
                   printf("Error on line %d: Parameter in call to function
\"%s\" has wrong type.\n", line_counter, entry->name);
                   exit(1);
                   }
               }
             }
           $$ = entry;
         }
       }
     ;
%%
int main(int argc, char *argv[])
{
  if ( argc > 0 )
    yyin = fopen( argv[1], "r" );
  else
    yyin = stdin;
  return yyparse();
}
```

This is the contents of the `datastructure.h` file. It defines the parts of the data structure used in the scanner/parser program and the functions that define the operations that can be done on these data structures.

```
#ifndef DATASTRUCTURE_H
#define DATASTRUCTURE_H

/*******************************/
/* The symbol table structure: */
/*******************************/

struct table_entry
{
  char* name;                       /* The name of the function or variable.
                                   Function names and variable names cannot
                                   overlap within the same scope. */
  double real_value;                /* Set to the variable's value if the
                                   variable is a real. */
  int bool_value;                   /* Set to the variable's value if the
                                 variable is a bool. */
  int type;                         /* Indicates the return type of a function
or
                                 the type of a variable. */
  int is_func;                      /* If set the node represents a funcion.
                                   If not, the node represents a variable.*/
  int has_param;                    /* Indicates if a function has a parameter
or not. */
  int param_type;                   /* The type of the function's parameter. */
  struct table_entry *left, *right; /* Syntax tree pointers. */
};

struct stack_entry
{
  struct table_entry* table_entries[100];
  int length;
  int free_index;
  struct stack_entry* below_element;
};

static struct stack_entry* top = NULL;
int teller = 0;

/*************************/
/* Symbol table methods: */
/*************************/

void addToTop(struct table_entry* t)
{
  if(top->free_index < 100)
    {
      top->table_entries[top->free_index] = t;
      top->free_index++;
    }
  else
    {
      printf("Error: Too many variables declared in scope. (Cannot exceed
100.)");
      exit(1);
    }
}
```

```
struct table_entry* getTableEntry(char* name)
{
  int i, l = top->length;
  struct stack_entry* current = top;
  char* n = "";
  while(current != NULL)
    {
      for(i = 0; i < l; i++)
      {
        if(current->table_entries[i]->name != NULL)
          {
            n = current->table_entries[i]->name;
            if(strcmp(name, n) == 0)
            return current->table_entries[i];
          }
      }
      current = current->below_element;
    }
  return NULL;
}

void push()
{
  // Just adds a new stack_entry struct to the top of the stack.
  int i;
  struct stack_entry *element;
  element = (struct stack_entry*)malloc(sizeof(struct stack_entry));
  element->length = 100;
  teller++;
  for(i = 0; i < 100; i++){
    element->table_entries[i] = (struct table_entry*)malloc(sizeof(struct
table_entry));
    element->table_entries[i]->name = NULL;
  }
  element->free_index = 0;
  if(top == NULL)
    element->below_element = NULL;
  else
    element->below_element = top;
  top = element;
}

void pop()
{
  struct stack_entry* old_top = top;
  top = old_top->below_element;
  free(old_top);
}

int existsInTop(char* name)
{
  int i, l = top->length;
  struct stack_entry* current = top;
  char* n = "";

  for(i = 0; i < l; i++)
    {
      if(current->table_entries[i]->name != NULL)
      {
        n = current->table_entries[i]->name;
        if(strcmp(name, n) == 0)
          return 1;
      }
      else
      return 0;
    }
  return 0;
}

int existsInStack(char* name)
{
  int i, l = top->length;
  struct stack_entry* current = top;
  char* n = "";
  while(current != NULL)
    {
      for(i = 0; i < l; i++)
      {
        if(current->table_entries[i]->name != NULL)
          {
            n = current->table_entries[i]->name;
            if(strcmp(name, n) == 0)
            return 1;
          }
      }
      current = current->below_element;
    }
  return 0;
}

int isEmpty()
{
  if(top == NULL)
    return 1;
  return 0;
}

void printStack()
{
  int i, n = 0, l = top->length;
  struct stack_entry* current;
  current = top;
  printf("\nSymbol stack: (from top to bottom)\n");
  while(current != NULL)
    {
      printf("free_index = %d, length = %d\n", current->free_index, current-
>length);
      printf("element %d = [", n);
      for(i = 0; i < l; i++)
      {
        if(current->table_entries[i]->name != NULL)
          printf("%s ", current->table_entries[i]->name);
      }
      printf("]\n");
      current = current->below_element;
      n++;
    }
  printf("\n");
}

void addName(struct table_entry* entry, int current_type)
{
  struct table_entry* temp;
  temp = (struct table_entry*)malloc(sizeof(struct table_entry));

  temp->name = entry->name;
  temp->type = current_type;
  temp->is_func = entry->is_func;
  temp->has_param = entry->has_param;
  temp->param_type = entry->param_type;

  if(temp->name != NULL)
    addToTop(temp);
}

/************************/
/* Syntax tree methods: */
/************************/

void setLeft(struct table_entry* n, struct table_entry* new_node)
{
  if(n->left != NULL)
    n->left = new_node;
  else
    {printf("n->left not NULL! Check C-code!\n"); exit(1);}
}

void setRight(struct table_entry* n, struct table_entry* new_node)
{
  if(n->right != NULL)
    n->right = new_node;
  else
    {printf("n->right not NULL! Check C-code!\n"); exit(1);}
}

void printTree(struct table_entry* node)
{
  if(node != NULL)
    {
      if(node->left != NULL)
      {
        printf("(");
        printTree(node->left);
        printf(")");
      }
      if(node->name != NULL)
      printf(" %s ", node->name);
      else
      {
        if(node->type == BOOL)
          {
            if(node->bool_value)
            printf("true");
            else
            printf("false");
          }
        else if(node->type == REAL)
          {
            printf("%.2f", node->real_value);
          }
      }
      if(node->right != NULL)
      {
        printf("(");
        printTree(node->right);
        printf(")");
      }
    }
  else
```

```
    printf("Error in printTree(): Node == NULL.\n");
}

#endif
```

This is the contents of the `tree.c` file. It defines the syntax tree structure used in the resulting scanner/parser program and the functions that define the operations that can be done on the nodes of the syntax tree.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

typedef enum {Var, Const, Add, Sub, Mul, Div, Pow, Relop} NodeType;
typedef struct node
{
  NodeType type; // Holds the sematic value of the node.
  char* string;  // The string that was found by lex.
  struct node *left = NULL, *right = NULL;
} TreeNode;
typedef TreeNode* syntax_tree;

void setLeft(struct node* n, struct node* new_node)
{
  if(n->left != NULL)
    n->left = new_node;
  else
    {printf("n->left not NULL! Check your C-code!"); exit(1);}
}

void setRight(struct node* n, struct node* new_node)
{
  if(n->right != NULL)
    n->right = new_node;
  else
    {printf("n->right not NULL! Check your C-code!"); exit(1);}
}

void printTree(struct node* root)
{
  struct node* n = root;
  if(n->type == Var || n->type == Const)
    printf("%s", n->string);
  else
    {
      printf("%s", n->string);
      if(n->left->type == Var || n->left->type == Const)
      printTree(n->left);
      else
      {
        printTree(n->left);
      }
      if(n->right->type == Var || n->right->type == Const)
      printTree(n->right);

    }
}
```

```
int main()
{
  return 0;
}
```

7 References

C++ Resources Network, The (2000). *Introduction: History of C++*. http://www.cplusplus.com/info/history.html

Digital Equipment Corporation (September 1999). *Welcome to the world of Modula-3!* http://research.compaq.com/SRC/modula-3/html/home.html

Ewing, G. (August 2002). *Pyrex: A language for writing Python extension modules.* http://nz.cosc.canterbury.ac.nz/~greg/python/Pyrex/version/Doc/About.html

Ghezzi, C. and Jazayeri, M. (1998). *Programming Language Concepts*. John Wiley & Sons, Inc. ISBN: 0-471-10426-4.

Giannini, M. and Keogh, J. (2001). *Windows Programming: Programmer's Notebook.* ISBN 0-13-027845-9.

GNOME Project, The (2003). *PyGTK: GTK+ for Python.* http://www.pygtk.org/

Hammond, M. and Robinson, A. (1st Edition January 2000). *Python Programming on Win32.* ISBN: 1-56592-621-8.

Hugunin, J. (June 1996). *Building Ariel - OpenGL GUI and Python to C.* http://www.python.org/workshops/1996-06/papers/hugunin.IPCIV.html

Johnson, L. (2002). *FXPy a Python binding for FOX.* http://fxpy.sourceforge.net/

Josuttis, N. M. (2001). *Object-Oriented Programming in C++.* ISBN 0-470-84399-3.

Krogdahl, S. (2004). *Beskrivelse av programmeringsspråket Pedagola: Til obligatorisk oppgave i INF 5110, våren 2004*. http://www.ifi.uio.no/inf310/oblig-sprak-2004

Langtangen, H. P. (August 2002). *Scripting Tools for Scientific Computations.*

Lewis, H. R. and Papadimitriou, C. H. (1998). *Elements of the Theory of Computation, Second Edition*. Prentice Hall, Inc. ISBN: 0-13-262478-8.

LinuxDevices.com (2005). http://www.linuxdevices.com

Louden, K. C. (1997). *Compiler Construction: Principles and Practice*. PWS Publishing Company, ISBN: 0-534-93972-4.

Lundh, F. (1999). *An Introduction to Tkinter.* http://www.pythonware.com/library/tkinter/introduction/

MacIntyre, B. (1997). *Modula-3 Reference and Tutorial.* http://www1.cs.columbia.edu/graphics/modula3/tutorial/.index.html

Pemberton, S. (May 2003). *A Short Introduction to the ABC Language.* http://homepages.cwi.nl/~steven/abc/

Rossum, Guido van, Drake, Fred L. (June 22, 2001). *Extending and Embedding the Python Interpreter*. http://www.python.org/doc/2.0.1/ext/ext.html

Python.org (May 20, 2004). *History of the software.* http://www.python.org/doc/2.3.4/ref/node104.html

Python.org (November 29, 2004). *Python 2.4. Documentation.*

Rempt, B. (January 1, 2002). *Gui Programming With Python: Using the Qt Toolkit.* Opendocs Llc, ISBN: 0970033044.

Smart, J., Roebling, R., Zeitlin, V., Dunn, R., et al (October 2004). *wxWidgets 2.5.3: A portable C++ and Python GUI toolkit.* http://www.wxpython.org

Strout, J. and Heise, D. (1998). *python2c* (Python source code)http://www.strout.net/python/ai/python2c.py