LANGUAGE IMPLEMENTATION BY SOURCE TRANSFORMATION

by

Pooja Dayanand

A thesis submitted to the
School of Computing
In conformity with the requirements for
the degree of Master of Science

Queen’s University
Kingston, Ontario, Canada
(January, 2008)

Copyright ©Pooja Dayanand, 2008
Abstract

Compilation involves transforming a high level language source program into an equivalent assembly or machine language program. Programming language implementation can therefore be viewed as a source to source transformation from the original high level source code to the corresponding low level assembly language source code. This thesis presents an experiment in implementing an entire programming language system using declarative source transformation. To this end a complete compiler/interpreter is implemented using TXL, a source transformation system. The TXL-based PT Pascal compiler/interpreter is implemented in phases similar to those in a traditional compiler. In the lexical and syntactic analysis phase any lexical and syntactic errors present are detected when the source program is parsed according to the TXL grammar specified. The semantic analysis phase is then run in which semantic checks are performed on the source program and error messages are generated when semantic errors are detected. The source program is also annotated with type information. The typed intermediate code produced by the semantic analysis phase can be directly executed in the execution phase. Alternatively, the intermediate typed source can be transformed into a bytecode instruction sequence by running the code generation phase. This bytecode instruction sequence is then executed by a TXL implementation of an abstract stack machine in the code simulation phase. The TXL-based PT Pascal compiler/interpreter is compared against the traditional S/SL implementation of the PT Pascal compiler. The declarative style of TXL makes the rules and functions in the TXL-based PT Pascal compiler/interpreter easier to understand and the number of lines of code in the TXL implementation is less than in the S/SL implementation. The TXL implementation is however slower and less scalable. The implementation of the TXL-based PT Pascal compiler/interpreter and the advantages and disadvantages of this approach are discussed in greater detail in this thesis.
Acknowledgements

I would like to first thank my supervisor Dr. J.R. Cordy for all his help and patience. I would also like to thank Manar Alalfi, Nevon Brake, Scott Grant, Adrian Thurston and Chanchal Roy for their suggestions. I am very grateful to my mother, my family and my friends for their support and encouragement.
# Table of Contents

Abstract............................................................................................................................................ ii  
Acknowledgements......................................................................................................................... iii  
Table of Contents............................................................................................................................ iv  
List of Listings ................................................................................................................................ vi  
List of Figures.................................................................................................................................. x  
List of Tables ................................................................................................................................. xii  
Chapter 1 Introduction ..................................................................................................................... 1  
  1.1 Motivations ............................................................................................................................ 1  
  1.2 Contributions ......................................................................................................................... 2  
  1.3 Overview of Chapters ............................................................................................................ 2  
Chapter 2 Background and Related Work ....................................................................................... 4  
  2.1 Compilers............................................................................................................................... 4  
  2.2 PT Pascal ................................................................................................................................ 5  
  2.3 PT Pascal Compiler ................................................................................................................ 6  
      2.3.1 S/SL ................................................................................................................................ 6  
      2.3.2 PT Pascal Compiler Phases ............................................................................................. 8  
  2.4 Source Transformation Systems .......................................................................................... 11  
      2.4.1 ASF+SDF...................................................................................................................... 11  
      2.4.2 Stratego ......................................................................................................................... 12  
      2.4.3 TXL............................................................................................................................... 13  
  2.5 Language Implementation by Source Transformation......................................................... 16  
      2.5.1 Tiger in Stratego ........................................................................................................... 16  
Chapter 3 Overview ....................................................................................................................... 18  
  3.1 Lexical and Syntactic Analysis Phase.................................................................................. 20  
  3.2 Semantic Analysis Phase ..................................................................................................... 21  
  3.3 Execution Phase ................................................................................................................... 22  
  3.4 Code Generation Phase ........................................................................................................ 24  
  3.5 Code Simulation Phase ........................................................................................................ 25  
Chapter 4 Lexical and Syntactic Analysis ..................................................................................... 27  
  4.1 Lexical Analysis................................................................................................................... 27  
Chapter 5 Semantic Analysis......................................................................................................... 39  
  5.1 Symbol Table ....................................................................................................................... 39
Chapter 5 

5.2 Implementation of Scopes .................................................................................................... 48
5.3 Symbol and Type Checking ................................................................................................. 53
5.4 Rewriting ............................................................................................................................. 61

Chapter 6 Execution Phase ............................................................................................................ 67

6.1 Memory ................................................................................................................................. 67
6.2 File Buffers ............................................................................................................................ 75
6.3 Statement Execution ............................................................................................................ 82
   6.3.1 Expression Evaluation .................................................................................................. 82
   6.3.2 Statements ..................................................................................................................... 85

Chapter 7 Code Generation and Simulation .................................................................................. 96

7.1 Grammar Overrides ............................................................................................................. 96
7.2 Variable Allocation .............................................................................................................. 97
7.3 Procedure Code Generation ............................................................................................... 99
7.4 Statement Code Generation ............................................................................................... 102
   7.4.1 Expression Code Generation ....................................................................................... 102
   7.4.2 Statement Code Generation ........................................................................................ 109

Chapter 8 Evaluation .................................................................................................................... 128

8.1 Qualitative Evaluation ....................................................................................................... 128
8.2 Quantitative Evaluation ..................................................................................................... 132

Chapter 9 Conclusion ................................................................................................................... 137

References .................................................................................................................................... 139
# List of Listings

2.1 Example S/SL program ................................................................. 8
2.2 Example ASF+SDF program .................................................... 12
2.3 Example TXL grammar .............................................................. 14
2.4 Example TXL rules ................................................................. 15
4.1 Keys statement in PT Pascal grammar ...................................... 28
4.2 Screening keywords in PT Pascal compiler ............................... 28
4.3 Compounds statement in PT Pascal grammar ......................... 29
4.4 Scanning a compound token in S/SL ....................................... 29
4.5 Comments statement in PT Pascal grammar ............................ 29
4.6 Scanning a comment in S/SL .................................................. 30
4.7 Tokens statement in PT Pascal grammar .................................. 30
4.8 Scanning an identifier in S/SL ................................................ 31
4.9 Program definition in PT Pascal grammar ................................ 31
4.10 S/SL specification of PT Pascal program .................................. 32
4.11 Program parameter definition in PT Pascal grammar ............... 33
4.12 S/SL specification of PT Pascal constants ............................... 33
4.13 Example of 'opt' modifier use in PT Pascal grammar ............... 34
4.14 Scanning optional subscript in S/SL ....................................... 34
4.15 Example of 'repeat' modifier use in PT Pascal grammar .......... 35
4.16 S/SL specification of a term in the PT Pascal compiler .......... 35
4.17 Example of sequence with one or more items in PT Pascal grammar ............................. 36
4.18 S/SL specification of variable declarations in PT Pascal compiler ........................................ 36
4.19 Example of 'list' modifier use in PT Pascal grammar ............... 36
4.20 S/SL specification of case labels in PT Pascal compiler ............ 37
4.21 Example of formatting nonterminal use in PT Pascal grammar .... 37
5.1 Grammar definition of the nonterminal [frame] ......................... 40
5.2 Grammar definition of nonterminal [symbol] ............................ 41
5.3 Implementation of symbol table in the TXL-based PT Pascal compiler/interpreter .......... 43
5.4 Implementation of symbol table in the PT Pascal compiler ........ 43
5.5 TXL rule that creates and adds symbol table entries for PT Pascal constants ....................... 44
5.6 TXL function that emits an error message if a symbol is doubly declared ......................... 45
5.7 TXL function that adds a new symbol entry to the symbol table ................................. 46
6.14 Recursive TXL function that skips leading white spaces ...................................................... 80
6.15 Recursive TXL function that collects non-blank characters ................................................... 81
6.16 TXL rule that resolves subscripts .......................................................................................... 83
6.17 TXL rule that replaces variable references with their values.................................................. 84
6.18 TXL rule that evaluates a PT Pascal expression ................................................................. 84
6.19 TXL function that executes an assignment statement........................................................... 85
6.20 TXL rule that removes procedures and adds them to a sequence ........................................ 85
6.21 TXL function that executes a call statement ......................................................................... 86
6.22 Recursive TXL function that executes a repeat until statement ............................................. 87
6.23 TXL function that executes a call to the PT Pascal procedure rewrite .................................... 89
6.24 TXL functions that evaluate write expressions and add them to write buffers .................... 90
6.25 TXL function that writes to an integer file ............................................................................ 91
6.26 TXL functions that execute a call to writeln with no argument ........................................... 92
6.27 TXL function that executes a call to the reset procedure ...................................................... 93
6.28 TXL function that executes a call to read a character ............................................................ 93
6.29 TXL functions that execute a call to the readln procedure .................................................. 94
7.1 TXL override of the definition for [declarations] in Code Generator ........................................ 97
7.2 TXL function that emits tcode sequence for a variable declaration ......................................... 98
7.3 TXL function that emits tcode instructions for a subrange declaration .................................... 98
7.4 TXL function that generates code for a procedure .................................................................. 102
7.5 TXL function that emits a tcode instruction for a literal character ....................................... 103
7.6 TXL function that emits tcode instructions for a string ........................................................... 103
7.7 TXL functions that generate fetch codes for variable references .......................................... 105
7.8 TXL functions that convert a term into postfix form and generate code for it ....................... 107
7.9 TXL functions that generate code for an expression ............................................................ 108
7.10 TXL function that generates code for an assignment statement .......................................... 110
7.11 TXL function that generates code for a while statement ...................................................... 113
7.12 TXL function that generates code for a call to read_char .................................................... 118
7.13 TXL main function in the Code Simulator that initialises the global variables ...................... 120
7.14 TXL grammar definition of [operand] ............................................................................... 121
7.15 TXL function in the Code Simulator that creates a memory cell for a subrange ................... 122
7.16 TXL function in the Code Simulator that creates a memory cell for a variable .................... 123
7.17 TXL function in the Code Simulator that creates a filebuffer ......................................... 123
7.18 TXL grammar definitions of [program] and [tCodeInstruction] ................................................. 124
7.19 TXL grammar definitions of [label] and [instruction] ............................................................... 124
7.20 TXL rule in the Code Simulator that executes the instructions ................................................ 125
7.21 TXL function in the Code Simulator that halts execution ......................................................... 126
8.1 Original grammar definition for if statement ........................................................................... 129
8.2 Additions to the grammar to permit elsif clauses ................................................................. 129
8.3 Original main function of the Parser ...................................................................................... 129
8.4 TXL rule added to the Parser to change elsif into nested if else .......................................... 130
8.5 Original S/SL rule that generates semantic tokens for an if statement .................................. 131
8.6 Modified S/SL rule that accepts elsif and emits nested if else semantic tokens ............. 131
**List of Figures**

2.1 Phase structure of the PT Pascal compiler ................................................................. 10
3.1 Data and control flow in the TXL-based PT Pascal compiler/interpreter .................... 19
3.2 Example of an error message emitted in the lexical and syntactic analysis phase .......... 20
3.3 Example of an error message emitted in the semantic analysis phase ......................... 21
3.4 Example of output from the semantic analysis phase ............................................... 22
3.5 Example of an error message emitted during the execution phase ............................. 23
3.6 Example of output from the execution phase ........................................................... 23
3.7 Example of output from the code generator ............................................................ 24
3.8 Example of output from the Code Simulator ............................................................ 25
3.9 Example of an error message generated by the Code Simulator ......................... 26
5.1 Example of a symbol table created by the Semantic Analyser ..................................... 42
5.2 Example of error message emitted when a variable is doubly declared ...................... 46
5.3 Example of scopes in the TXL implementation of the symbol table ......................... 50
5.4 Example of symbol checking done by the Semantic Analyser .................................. 54
5.5 Example of error message generated by the expression checker module ................. 56
5.6 Example of evaluating the type of a term ............................................................... 60
5.7 Example of transformed read and write calls .......................................................... 63
6.1 Example of the Memory in the execution phase ....................................................... 69
6.2 Example of error message produced when an uninitiated variable is referenced ........ 71
6.3 Example of error message produced when a value is outside subrange bounds ........... 73
6.4 Scopes as implemented in the execution phase ....................................................... 75
6.5 Example of error message emitted when end of file is reached .................................. 82
6.6 Example of error message emitted if the case selector does not match any case labels 89
7.1 Example of tcode instructions emitted for variable declarations ............................. 99
7.2 Example of code that is generated for a procedure ................................................. 101
7.3 Example of code generated for variable references and literals in expressions .......... 106
7.4 An example of the code generated for an expression .............................................. 109
7.5 Example of code generated for a call statement ..................................................... 111
7.6 Example of the code generated for an if statement ................................................. 112
7.7 Example of the code generated for a while statement ............................................. 114
7.8 Example of the code generated for a case statement .............................................. 116
7.9 Example of code generated for calls to write and write_int .................................. 117
7.10 Code generated for a writeln statement

119
List of Tables

8.1 Description of test programs ........................................................................................................ 133
8.2 Comparison of TXL-based PT Pascal compiler/interpreter and traditional PT Pascal compiler execution times ......................................................................................................................... 134
8.3 Number of lines of code in the TXL-based PT Pascal compiler/interpreter and traditional PT Pascal compiler ................................................................................................................................ 135
Chapter 1

Introduction

TXL [Cordy06] was created in 1985 as a rapid prototyping system to support experiments in language design. It was initially intended as a tool to prototype extensions to the Turing language. It has since grown into an all purpose transformation system whose application has spread beyond language design to areas such as software engineering [Cordy02], pattern recognition [Kiy05] and artificial intelligence [Zan02]. The purpose of this thesis is to add to this growing body of work involving TXL by experimenting with implementing a programming language entirely in TXL.

1.1 Motivations

Compilation is defined as the translation of a source program in a high level language into an equivalent program in a low level language like assembly or machine language. Source transformation systems are well suited to translation tasks. This thesis describes the implementation of the PT Pascal [Ross80] language using the TXL source transformation system. The main motivation behind this experiment was to explore whether it was possible to implement the traditional phase structure of the compiler using TXL. By preserving the traditional structure, the TXL-based PT Pascal compiler/interpreter can be used to teach compiler implementation at an introductory level while at the same time introducing a source transformation technology to the students. The thesis also explores how the differences between the traditional implementation of the PT Pascal compiler and the TXL-based PT Pascal compiler/interpreter makes it easier to demonstrate important compiler concepts while hiding complex implementation details.

This experiment also allows us to explore the advantages and challenges involved in implementing a language using a source transformation system. The rewriting capabilities of source transformation systems in general and the by-example nature of TXL in particular facilitate certain tasks. Moreover TXL is a self contained system that performs parsing, pattern matching, transformation and unparsing without requiring any additional tools. This eliminates the added complexity of using different tools during different stages in the compilation process as is sometimes done in the traditional implementation of compilers. The advantages as well as some of the challenges involved in implementing a language by source transformation, using TXL as an example, are discussed in greater detail in the chapters that follow. Another motivation for this work is to create a fully worked out example of language implementation using TXL. TXL has a
growing community of users and it is important to have different applications of TXL available in the public domain to help new users learn the technology.

1.2 Contributions

In the course of this experiment two complete interpreters have been implemented for the PT Pascal language using TXL. The first interpreter is an implementation of the modern approach to interpretation and directly interprets a PT Pascal source program that has been annotated with type information. The second interpreter is a byte code interpreter and interprets the intermediate code generated for the PT Pascal source program. This TXL-based implementation of the PT Pascal language demonstrates important compiler implementation concepts such as symbol tables, scope implementation and code generation in an easy to understand manner making it useful as a teaching tool. It also serves as a fully worked out example of language implementation using TXL and illustrates various capabilities of the TXL source transformation system.

1.3 Overview of Chapters

Compiler construction is a very mature science and the traditional structure of a compiler is well defined and modular. The TXL-based PT Pascal compiler/interpreter adheres to this structure which is described in detail in Chapter 2. An overview of the PT Pascal language itself as well as a description of the existing implementation, using traditional technology, of the PT Pascal compiler is provided in this chapter. A survey of existing source transformation systems and language implementations using these systems is also included in Chapter 2 along with a detailed description of TXL.

Chapter 3 contains an overview of the TXL-based PT Pascal compiler/interpreter. The phases in the TXL-based PT Pascal compiler/interpreter as well as the components that make up each of the phases are described and examples are given of the inputs into and the output from each phase. This chapter also discusses the relationship between the phases. The lexical and syntactic analysis phase is described in detail in Chapter 4 along with relevant code listings and examples. Comparisons are also drawn between the S/SL [Holt82] implementation of the phase in the traditional implementation of the PT Pascal compiler and the TXL implementation described in this thesis. Chapter 5 contains a description of the semantic analysis phase. The structure and implementation of the symbol table and detailed discussions of the various analysis tasks carried out in this phase are included in this chapter.
Chapter 6 discusses the implementation of the execution phase in which the intermediate source program produced by the semantic analysis phase is executed. This chapter includes detailed descriptions of how the memory is simulated and the implementation of I/O operations. The code generation phase which converts the intermediate source from the semantic phase into an abstract machine code called tcode is discussed in Chapter 7. This chapter also contains the details of the code simulator that was created to execute the tcode instruction sequence produced by the code generation phase. Chapter 8 concludes this thesis by summarising the work described here and discussing directions for future work.

The next chapter provides background information on the traditional structure of programming language compilers, the PT Pascal language, source transformation systems and previous work on language implementation using source transformation systems.
Chapter 2

Background and Related Work

The first chapter explained the motivations behind the work done in this thesis. In this chapter the background information and some related work done in the area are briefly detailed. The traditional phase structure of compilers is first discussed followed by a description the PT Pascal language and the structure of the existing PT Pascal compiler. Some source transformation systems are then reviewed followed by an overview of the TXL transformation system. Language implementation by source transformation is also discussed with a brief look at the implementation of the Tiger language [Appel98] in Stratego [Vis01, Vis04].

2.1 Compilers

A source program in a high level language must be translated into assembly or machine language before it can be executed [Cordy01]. A program that performs this translation is called a compiler. Traditionally the task of compilation is divided into four phases. These phases can be run in sequence with the output from one phase used as the input into the phase that follows or concurrently as co-routines.

The first phase in a compiler is the lexical analysis phase. In this phase the text of the source program is converted into language tokens such as identifiers, integers and operators. Keywords are also identified in this phase. Any characters in the input text that are not legal characters in the language being compiled result in error messages being emitted.

The lexical analysis phase is followed by the syntactic analysis phase. The token stream produced by the lexical analysis phase is checked to ensure that the source program adheres to the syntactic rules of the language. Any syntax errors that are detected result in error messages being emitted. The input token stream is converted into a postfix token stream or a parse tree in the process.

The output from the syntactic analysis phase is then used as input into the semantic analysis phase. In this phase various checks are carried out to ensure that the source program is legal given the semantic constraints of the language. This is done by creating a symbol table that contains the attribute information of the symbols in the source program and then using this stored information to perform tasks such as symbol checking, type checking and scope analysis. Error messages are
emitted if any semantic errors are found in the source program. The output from the semantic analysis phase is an annotated parse tree or an intermediate code.

In the final phase of compilation assembly or machine code is generated for the target machine based on the output from the semantic analysis phase. In the code generation phase runtime errors such as array indices being out of bounds and uninitiated variables being referenced are detected and appropriate error messages are emitted. The code that is generated can also be optimised in this phase. An example of a compiler with the traditional phase structure is shown in Figure 2.1.

Alternatively, the intermediate code produced by the semantic analysis phase can be directly executed by an interpreter instead of being converted into machine code.

2.2 PT Pascal

PT Pascal [Ross80], the language implemented in this experiment, is a subset of Standard Pascal [Addy80] that was created by J. Alan Rosselet. Integer and string constants can be defined in PT Pascal. It has a set of predefined simple data types: integer, char, Boolean, text and subrange. The available complex data types are array and file. The component type of an array must be a simple type and file components can only be of the type text, char or integer. User can also define types based on the predefined types. Variables are declared as being of either a predefined or user defined type. PT Pascal also allows procedures with both value and variable parameters.

Assignments in PT Pascal are scalar. Conditional statements can be in the form of if statements with optional else clauses or case statements. Call statements, while loops and repeat until loops are also allowed. I/O operations are performed using a set of predefined procedures. A call to the rest procedure opens a file for reading. The read procedure reads a character or integer from the standard input stream or file into a variable listed as an argument. A call to the readln statement results in a newline character being read from the standard input stream or the file. The rewrite procedure opens a file for writing. Expressions can be written to the standard output stream or a file by making a call to the write procedure. The writeln procedure writes a newline character to the standard output stream or a file.

PT Pascal also has a few predefined functions. The ord function takes a character as an argument and returns the numeric value that corresponds to the character in the character set of the target machine. The chr function does the opposite where it accepts an integer as argument and then
returns the corresponding character from the character set. The eoln and eof functions return true if it is the end of the line being read or end of the file being read respectively. PT Pascal also allows external procedures which are procedures that have been separately compiled. This feature has not been implemented in the TXL-based PT Pascal compiler/interpreter described in this document.

2.3 PT Pascal Compiler

The traditional implementation of the PT Pascal compiler [Ross80] is bootstrapped and consists of four phases as described earlier. Each of the phases use table driven logic which is programmed in S/SL [Holt82].

2.3.1 S/SL

S/SL (Syntax/Semantic Language) is a special-purpose control language that is used to implement compilers. It is a data-free language which makes use of semantic mechanisms to indirectly manipulate data. Semantic mechanisms are modules that contain semantic operations that access and manipulate the data. An interface to the semantic mechanism is specified in S/SL and the semantic operations that are defined can be invoked in the S/SL program when data handling is required. These semantic operations are however implemented in separately in a base language.

An S/SL program consists of definitions of values used in the S/SL program, specifications of the interface to semantic mechanisms and a set of rules. The values used in S/SL program are the input tokens, output tokens, error signals and other user defined values. Each of these values in these sets is given a name. In the case of input and output tokens a string name can also be associated with the name of the token.

Semantic mechanisms are specified in an S/SL program by enumerating the semantic operations that provide an interface to the mechanism after the name of the mechanism. Each rule in an S/SL program consists of a sequence of actions. There are eight actions in S/SL. The call action is used to call an S/SL rule and is represented by a ‘@’ character followed by the name of the rule that is being called. The return action is used to return a value from a rule and is indicated by a ‘>>’ symbol followed by the value to be returned. The input action, indicated by the presence of an input token in a rule, means that the next token in the input stream must match the input token in the rule. The character ‘?’ can be used to match any token in S/SL.
The emit action represented by the character ‘.’ followed by an output token indicates that the
token is to be emitted to the output stream. The error action consists of a ‘#’ character followed by
an error signal and indicates that an error message should be emitted to the error stream. The
cycle action is represented by a sequence of actions enclosed in braces ‘{ }’ and means that the
enclosed actions are to be repeated until a return symbol ‘>>’ or a cycle exit symbol ‘>’ is
encountered.

An S/SL choice action is used to select a set of actions to perform based upon the value of an
optional selector. Each choice alternative is marked by a list of one or more labels and contains
zero or more actions. If not selector is specified then an alternative is selected by trying to match
the current input token with a choice label. If the selector is a call to a rule, the rule is executed
and an alternative is chosen based on the return value from the rule. The name of a semantic
operation can also be used as a selector in which case the value returned by the semantic
operation is used to select an alternative. The default alternative is denoted as having the character
‘*’ for a label and is matched if no other label in the choice action is matched.

The syntax of a choice action is as follows:

```
[selector
  | labels:
    actions
  ...
  | *:
    actions
]
```

Listing 2.1 shows an example of an S/SL program based on an example in ‘An Introduction to
S/SL’ [Holt82] that scans simple expressions, with integers and the operators ‘+’ and ‘-’,
separated by semicolons. If blank spaces are ignored and any other characters are considered
illegal and result in error signals being emitted if found in the input stream.
2.3.2 PT Pascal Compiler Phases

The lexical analysis phase in the PT Pascal compiler is implemented as two routines the scanner routine and the screener routine. The logic for this phase is specified using S/SL. The S/SL tables that are created are used to drive these routines that are implemented in PT Pascal. The scanner accepts the text of the source program character by character and converts it into language tokens.

```
input:
  digit
  blank
  illegalChar;
output:
  integer;
input output:
  semicolon ';'
  plus    '+'
  minus   '-';
error:
  badChar;
mechanism Buffer:
  BufferSave; % semantic operation that saves the
              % last character that was accepted
rules
Scan:
{
  | digit:
      BufferSave
      {
        | digit:
            BufferSave
            |
            +:
            |
            -:
            |
            blank:
            |
            illegalChar:
              #badChar
     >
  } |
  semicolon
  |
  plus
  |
  minus
  |
  blank:
}};
end
```

Listing 2.1: Example S/SL program
such as identifiers, integers, literals, operators and other character tokens such as braces and
semicolons while removing white spaces and comments.

The scanner is unable to distinguish between keywords and user defined identifiers and emits both
of them as identifier tokens. The output tokens produced by the scanner routine are then run
through the screener routine which identifies keywords and emits the appropriate keyword tokens
in their place. It places the identifiers in identifier table and replaces each of them with the index
of their table entries. It also replaces strings of digits with the integer values that they represent.

Syntactic analysis is done by the parser routine. The parser is run as a co-routine with the scanner
and screener routines so both lexical and syntactic analyses are done in one pass. The parser
accepts the tokens produced by the scanner/screener as input and checks for syntax errors. The
S/SL program that drives the parser accepts each lexical token and converts the token stream into
a postfix token stream while replacing the parse token with an equivalent semantic token. A
syntax error is discovered when a token in the input stream does not match the expected character
that is specified in the S/SL program in which case an error signal is emitted. When a syntax error
is detected the parser goes into error recovery mode in which all parse tokens until the next
statement separator (a semicolon in the case of PT Pascal) are accepted but no semantic tokens are
emitted. After the statement separator token is accepted parsing continues normally.

In the semantic analysis phase various semantic checks are performed on the token stream that is
output by the parser, code locations are assigned to the variables and tcode instructions are
emitted. The S/SL program in the semantic analysis phase makes use of various semantic
mechanisms in order to manipulate the data required by the semantic analysis tasks. The symbol
table and symbol stack mechanisms are used to collect, store and lookup the attributes of symbols
declared in the source program. The type table and type stack mechanisms are used to store type
information and to evaluate the types of expressions when type checking is performed.

Other mechanisms such as the storage allocation mechanism, the emit mechanism and the fix
address stack mechanism are used to assist with the emission of t-code instructions. The storage
allocation mechanism generates the data area addresses and the code area addresses required for
the data objects and the code that is emitted. The emit mechanism is used to emit the operands
required by the tcode instructions. The fix address stack mechanism is used to save the branch
addresses required when emitting code for if, case, while and repeat until statements.
In the code generation phase the tcode stream produced by the semantic analysis phase is used to generate assembly code for the target machine. The S/SL program in the coder uses the operand stack mechanism to process expressions. It also uses certain other mechanisms that are analogous to the ones used in the semantic analysis phase such as the fix address stack mechanism and the emit mechanism. The coder performs certain local optimizations on the code it generates such as factoring out the constants, replacing multiplication by powers of two with left shift operations, using machine idioms such increment and decrement and replacing expressions with their values if the values of all the operands are available at compile time.

The PT Pascal compiler makes use of a runtime monitor to handle array subscripting and I/O operations. The runtime monitor is a collection of machine dependent support routines. Figure 2.1 taken from ‘Introduction to Compiler Construction using S/SL’ [Cordy01] shows the phase structure of the PT Pascal compiler along with the input to and output from each phase.

![Phase structure of the PT Pascal compiler](image)

Figure 2.1: Phase structure of the PT Pascal compiler
2.4 Source Transformation Systems

Source transformation systems are tools designed to assist in transforming one source text to another. These transformations are accomplished by first parsing the input source text to create an internal representation of it. The transformation engine then applies user defined rules that specify the intended transformation to this internal representation resulting in a transformed source text which is then written out as the output from the transformation. Some source transformation systems including TXL are discussed in this section.

2.4.1 ASF+SDF

ASF+SDF [Bran02] is a source transformation system that makes use of the term rewriting paradigm to perform source code analysis and source transformations. Term rewriting involves the transformation of an initial term into a simplified term through the repeated application of simplification rules. The ASF (Algebraic Specification Formalism) component of ASF+SDF is used to describe the semantic of languages with the help of rewrite rules. The SDF (Syntax Definition Formalism) component is used to describe the lexical and context-free syntax of the language.

The input is parsed according to the grammar defined using SDF and is then transformed by applying the rewriting rules specified in ASF to it. The lexical syntax in SDF describes the lexical tokens in the language while the context free syntax describes the syntactic structure of the language. Both the lexical syntax and context free syntax are specified as a set of productions. Each production consists of a set of one or more symbols on the left hand side and a single symbol on the right hand side separated by the operator “->”. The symbols on the left hand side describe the syntax of the symbol on the right hand side which is the type.

The rewriting rules specified in ASF are of the form LHS = RHS where both the LHS and RHS are terms. Any subterms in the input that the rule is applied to which match the left hand side are replaced by the right hand side. A variable can be bound in the left hand side and represents the term it is bound to when it appears on the left hand side. When it is present on the right hand side the term to which it is bound is copied into the replacement. Listing 2.2 shows an example of an ASF+SDF program, based on an example provided in a tutorial on ASF+SDF [ASF07], that resolves simple boolean expressions.
Stratego [Vis01, Vis04] is a source transformation system that is based on the term rewriting paradigm used by ASF+SDF. Stratego however introduces the separation of rewriting strategies from rewriting rules which allows the user to have greater control over the manner in which the transformation rules are applied. A Stratego program consists of an abstract syntax, a set of transformation rules and a set of strategies for applying those rules.

The abstract syntax of the input language is specified using SDF. The rewriting rules in the Stratego program are basic labelled rewrite rules of the form L: LHS -> RHS where the LHS and RHS are both terms and result in a pattern that matches the LHS being replaced with the RHS term.

Listing 2.2: Example ASF+SDF program

2.4.2 Stratego

Stratego [Vis01, Vis04] is a source transformation system that is based on the term rewriting paradigm used by ASF+SDF. Stratego however introduces the separation of rewriting strategies from rewriting rules which allows the user to have greater control over the manner in which the transformation rules are applied. A Stratego program consists of an abstract syntax, a set of transformation rules and a set of strategies for applying those rules.

The abstract syntax of the input language is specified using SDF. The rewriting rules in the Stratego program are basic labelled rewrite rules of the form L: LHS -> RHS where the LHS and RHS are both terms and result in a pattern that matches the LHS being replaced with the RHS term.
A strategy is defined as an operation that performs a transformation or fails. A rewrite rule is therefore the most basic strategy. More complex strategies can be specified in Stratego by combining basic strategies with strategy operators. There are two classes of strategy operators: operators for sequential programming and operators for term traversal. Strategies that apply to the root of a term are combined using sequential programming operators some examples of which are the identity operator (id) which always succeeds and returns the same term as the result, the sequential composition operator (s1; s2) which applies the strategy s1 followed by the strategy s2 and the choice operator (s1 + s2) which applies any one of the strategies s1 or s2.

The term traversal operators are used to when rules need to be applied throughout a term and not just at the root. Some term traversal operators are all(s) which applies the strategy s to all subterms in the current term, the try strategy which is defined as try(s) = s <+ id applies the strategy s to the current term or the strategy id if s fails and the repeat strategy, repeat(s) = try(s; repeat(s)) which repeatedly applies the strategy s until s fails.

2.4.3 TXL

TXL [Cordy05, Cordy06] is a first order functional and rule based language that is used to perform source transformations. The input to a TXL program is parsed based on the grammar specified y the user to create a program tree. The TXL rules specified by the user are then applied to the tree and the tree is transformed. This transformed tree is then unparsed using the same grammar to provide the transformed output.

A TXL program has two components. A user defined grammar and a set of transformation rules. The user defined grammar is further divided into a base grammar and an optional set of grammar overrides. A TXL grammar specifies the lexical and syntactic forms of the input language. The nonterminals defined in the TXL grammar form the structured types that are used to identify subtrees in the TXL program. Each nonterminal is defined as a sequence of terminal and nonterminal types. The lexical tokens of the input language form the terminal types in the TXL grammar. TXL has a set of predefined terminal types such as [id] and [number] but these definitions can be overridden in the TXL grammar and user defined terminal types can also be created. A TXL grammar is specified using BNF like notation. TXL grammar overrides are used to either extend or redefine the nonterminals already defined in the base grammar.
A TXL rule consists of a pattern and a replacement. The pattern is an example of the instance that must be matched and consists of tokens and variables. Variables can be bound to typed subtrees in the match. The replacement is an example of the transformed result and consists of tokens and references to variables. TXL rules specify the type of the subtree to be transformed and both the pattern and the replacement must be of the type specified for the rule to succeed. A TXL rule recursively matches all the instances of its pattern within its scope and replaces them with its replacement. The rule halts when it can no longer find a match within its scope.

TXL provides different types of transformation rules that can be used to control the manner in which the rule is applied to a tree. If each subtree in the scope is to be matched only once a one pass rule can be used. In a one pass rule the replace keyword is followed by a ‘$’ character. If a rule is to match only one instance of its pattern a TXL function is used instead. A TXL function has the same syntax as a TXL rule with the only difference being that the keyword rule is replaced by the keyword function. A function does not search so a match is only found if its whole scope matches the specified pattern. If the subtree is to be searched for a match a ‘*’ character is placed following the replace keyword. In this case the transformation is applied to the first matching subtree in the scope of the function. Conditions can be specified in rules and functions using the where clause.

A bound variable in TXL can be broken down into its constituent subtrees using a TXL deconstructor. The deconstruct of a variable succeeds if the pattern the subtree bound to the variable matches the pattern specified in the deconstructor. The typed subtrees in the
deconstructor pattern are also bound to variables. If the variable being deconstructed does not match the pattern specified in the deconstructor the deconstruct fails and causes the rule or function in which it is present to also fail. Deconstructors can therefore be used to create checks and halt rules or functions. A new bound variable can be created by specifying the pattern of the subtree to be bound to the variable in a TXL constructor. An example of a TXL program is shown in the Listings 2.3 and 2.4. Listing 2.3 shows the TXL grammar and Listing 2.4 shows the TXL rules.

```
rule main
  replace [program]
    P [program]
  deconstruct not P
    BCon [boolCon]
  by
    P [resolveTrueAnd] [resolveFalseAnd]
      [resolveTrueOr] [resolveFalseOr]
end rule

rule resolveTrueAnd
  replace [boolean]
    true & Bool [boolean]
  by
    Bool
end rule

rule resolveFalseAnd
  replace [boolean]
    false & Bool [boolean]
  by
    false
end rule

rule resolveTrueOr
  replace [boolean]
    true ’| Bool [boolean]
  by
    true
end rule

rule resolveFalseOr
  replace [boolean]
    false ’| Bool [boolean]
  by
    Bool
end rule
```

Listing 2.4: Example TXL rules
2.5 Language Implementation by Source Transformation

Compilation can be viewed as a source transformation task in which the source program is transformed in an equivalent assembly code sequence for the target machine. The semantic checks that need to be performed during compilation are however not typical of the tasks performed using source transformation systems. It is therefore an interesting experiment to implement a language using source transformations.

2.5.1 Tiger in Stratego

Tiger [Appel98] is a small imperative language defined by Andrew Appel in his book on compiler implementation. The Tiger language permits integer, string, array and record data types. A program in Tiger is a sequence of declarations and expressions. The declarations can be of variables, functions and types and can be nested. The language permits assignment expressions which do not yield values, if expressions with optional else clauses, while expressions and for expressions and function calls. It also has let expressions that contain declarations and expressions and represent a new scope. A compiler has been implemented in Stratego using rewrite strategies for the Tiger language [Tiger07].

The components that comprise the Tiger compiler are divided into three groups based upon the tasks they perform; the front end, the optimizer and the back end. The tasks accomplished by the components in these groups are described in the order in which they are performed.

In the front end of the Tiger compiler the syntax of the language is defined using SDF. This syntax definition is then used to generate a parse table. The abstract syntax of Tiger is also automatically derived from its syntax definition. The abstract syntax trees are then desugared. In the case of the Tiger compiler desugaring the abstract syntax trees involves representing operators in a general form using constructors and splitting let expressions so that each let expression only declares one variable, collection of functions or collection of types. The desugared abstract syntax can be evaluated by an interpreter defined for the purpose in the Tiger compiler. If the program is to be compiled instead of being interpreted the desugared abstract syntax is type checked and the variables are annotated with their types. Bound variables are then uniquely renamed so that variables in different scopes do not have the same names.

The annotated and renamed abstract syntax is optimised in the optimizer. Various optimizations such as constant folding, constant propagation and dead code elimination are applied to the
abstract syntax. These trees are then normalised to so that the expressions do not contain side
effects and are converted into a language independent internal representation.

In the back end of the Tiger compiler the internal representation is transformed into MIPS
assembly code by applying rewrite strategies that perform instruction selection and register
allocation. The assembly code generated is also optimised and is then executed on using the spin
simulator. The Tiger compiler also contains a number of utilities that pretty print and check the
format of the output from various transformations performed over the course of the compilation.

TXL, being a hybrid between a rule based and first order functional language and strongly typed
is different from Stratego in many respects. It therefore seems likely that implementing a
language using TXL will have different advantages and challenges. In the chapters that follow an
overview is given of the TXL-based PT Pascal compiler/interpreter and the each of the phases in
the interpreter is discussed in detail.
Chapter 3
Overview

Our TXL-based PT Pascal compiler/interpreter is implemented as a series of distinct phases. These phases are based on the traditional structure of compilers discussed in chapter 2 and each phase is run in a single pass. The first phase is the lexical and syntactic analysis phase. In this phase the program source is broken up into language tokens and then parsed according to the TXL [Cordy05] base grammar specified for PT Pascal [Ross80]. In the traditional compiler structure lexical and syntactic analyses are two separate phases. However in the TXL implementation lexical and syntactic analyses are implemented in a single phase.

The second phase is the semantic analysis phase where a symbol table is constructed and semantic checks are performed on the program source which is then converted into a typed intermediate source. After the semantic analysis pass a choice must be made before interpretation can proceed. The typed source produced by the semantic analysis phase can either be directly executed using TXL or it can be converted in a tcode instruction sequence and then executed by an abstract machine.

If the typed source is to be directly executed the execution pass is run and the output from this phase is the output of the source program. If, on the other hand, the source program is to be converted into a tcode instruction sequence the code generation pass is run. The tcode instruction sequence produced is then executed in the code simulation pass. These phases and their inputs and outputs are described in greater detail in the sections that follow. Figure 3.1 shows the data flow and the control flow in the TXL-based PT Pascal compiler/interpreter. Input and output artefacts are represented by rectangles and the rounded rectangles represent the phases.
Figure 3.1: Data and control flow in the TXL-based PT Pascal compiler/interpreter
3.1 Lexical and Syntactic Analysis Phase

The first phase in the TXL-based PT Pascal compiler/interpreter is the lexical and syntactic analysis phase. The PT Pascal source program is taken in as input to this phase. Lexical or syntactic errors are detected and error messages are emitted to the standard error stream. An example of the error messages emitted during this phase is shown in Figure 3.2. In traditional implementations of compilers the syntactic analysis phase produces a postfix token stream or a parse which is then used as input into the semantic analysis phase that follows. The Lexical and Syntactic Analyser in the TXL implementation however makes no changes to the input source program since the TXL engine will convert the input source program into a parse tree before applying any rules to it in the semantic analysis phase.

![Error Message](image)

Figure 3.2: Example of an error message emitted in the lexical and syntactic analysis phase

The lexical and syntactic analyses can be performed in the same phase as the semantic analysis. The advantage of splitting them into two separate phases is that TXL grammar overrides can be used to make changes to the structure of the parse tree of the input source program in the semantic analysis phase since the input grammar specified for this phase need no longer obey the syntactic constraints of the language. The increased flexibility of the input grammar increases the ease with which rules can be formulated to perform semantic analysis tasks. To ensure that a source program with lexical or syntactic errors is not executed the next phase is not run if any errors are detected at this point.
The Lexical and Syntactic Analyser consists of a TXL base grammar that specifies the syntax of PT Pascal and a TXL main function that matches the input source program.

### 3.2 Semantic Analysis Phase

The semantic analysis phase follows the lexical and syntactic analysis phase. This phase also takes the PT Pascal source program as its input. Any semantic errors detected by the semantic analyser result in error messages that are emitted to the standard error stream. Figure 3.3 shows an example of the error messages emitted in this phase.

```plaintext
program myprog
(output);
var
  i : integer;
begin
  i := 'p'
end.
```

Figure 3.3: Example of an error message emitted in the semantic analysis phase

The output from the semantic analyser is an intermediate version of the source code where all the variable references are replaced by their lexical level order number addresses. They are also annotated with their type and kind as this information is required by the code generator. Read and write calls with multiple arguments are turned into a sequence of calls with a file argument and a single read variable or write expression. The calls to the read and write procedures are also changed to reflect the types of the arguments in the calls. An example of the typed intermediate code generated by the Semantic Analyser can be seen in Figure 3.4.
Figure 3.4: Example of output from the semantic analysis phase

The Semantic Analyser consists of a TXL base grammar for PT Pascal, a set of grammar overrides and rules that construct a symbol table and perform semantic checks on the source program. The base grammar specifies the exact syntax of PT Pascal which is essential for the syntax checking done by the Lexical and Semantic Analyser but no longer necessary during the semantic analysis phase. The grammar overrides are used to change the parse tree of the input source program in ways that make it easier to create rules that perform the semantic checks.

The transformation rules are divided into modules based on their purpose. The main module contains the rules that populate the symbol table. The symbol table module contains a subgrammar that specifies the structure of the symbol table and rules that manipulate it. Other modules contain rules that evaluate expression types, perform semantic checks on statements, generate error messages and perform rewriting tasks.

### 3.3 Execution Phase

If the intermediate source program, produced as output from the semantic analysis phase, is to be directly executed the execution phase is run. The intermediate code produced by the semantic analysis phase is used as input to this phase after unnecessary annotations are removed. The interpreter detects runtime errors such as an array index being out of bounds and emits
appropriate error messages to the standard error stream. An example of an error message generated during the execution of a PT Pascal program can be seen in Figure 3.5. If a runtime error is encountered execution is halted. During execution user input is obtained from the standard input stream or files created by the user.

Figure 3.5: Example of an error message emitted during the execution phase

The output from this phase is the output produced by executing the source program as shown in Figure 3.6. The output is also written to the standard error stream.

Figure 3.6: Example of output from the execution phase

The execution phase consists of the same base grammar as is used in the previous phases along with a different set of grammar overrides. The execution rules are separated into two modules one of which executes the statements in the source program and the other module that evaluates
expressions. The structure of the memory and file buffers used in this phase and the rules that manipulate them are present in the memory and file buffer modules respectively.

### 3.4 Code Generation Phase

The code generation phase is run after the semantic analysis phase if the intermediate source is to be transformed into a sequence of tc ode instructions. The Code Generator takes the intermediate code produced by the Semantic Analyser as input. It emits no error messages and replaces the declarations and statements in the intermediate code with their equivalent tc ode instruction sequences. Each tc ode instruction consists of a numerical label that indicates its position in the instruction sequence, a tc ode and an optional sequence of operands as can be seen in the example in Figure 3.7.

```plaintext
program myprog (output);
  var
    % (0, 0) i : integer;
  begin
    % (0, 0) i Variable integer := 2 + 3;
    write (output, % (0, 0) i Variable integer);
  end.
```

![Figure 3.7: Example of output from the code generator](image-url)
The Code Generator makes use of the PT Pascal base grammar along with grammar overrides that specify the form of tcode instructions. The rules in the expression code generator module convert expressions in the source program into a postfix sequence of tcode instructions. The rules that create the tcode instructions that replace the declarations and statements in the source program with tcode instructions are present in the main module.

### 3.5 Code Simulation Phase

The tcode instruction sequence produced by the code generation phase is executed by the Code Simulator. The variable references in the tcode instructions retain the annotations made to them in the semantic analysis phase. Since these annotations are not used by the Code Simulator they are removed from the tcode instruction sequence before it is used as input to the code simulator. The output from the Code Simulator is the output produced by executing the PT Pascal source program. The code simulator gets user input in the same way that the interpreter does. An example of the output from executing a PT Pascal source program is shown in Figure 3.8.

```plaintext
0 : tLiteralAddress output
1 : tFileDescriptor
2 : tWriteBegin
3 : tLiteralString 12 'Hello world!'
4 : tStringDescriptor 3
5 : tLiteralInteger 1
6 : tLiteralAddress output
7 : tTrap trWriteChar
8 : tPopStack
9 : tPopStack
10 : tParmEnd
11 : tLiteralAddress output
12 : tTrap trWriteln
13 : tTrap trHalt
```

![Figure 3.8: Example of output from the Code Simulator](image-url)
During this phase runtime errors are detected and appropriate messages are generated. Execution is halted if an error occurs. An example of an error message emitted is shown in Figure 3.9.

The Code Simulator consists of a base grammar that specifies the syntax of the tcode instructions generated by the Code Generator and a main module that contains the rules that simulate the execution of the tcode instructions. It makes use of the same memory and file buffer systems specified in the memory and file buffer modules that are used by the execution phase.

The chapters that follow discuss the implementation of these phases of the TXL-based PT Pascal compiler/interpreter, starting with the lexical and syntactic analysis phase, in much greater detail and contain code listings that serve as examples of how various compilation tasks can be accomplished using TXL.
Chapter 4
Lexical and Syntactic Analysis

The overview provided in Chapter 3 described the phases of the PT Pascal [Ross80] interpreter. This chapter describes the lexical and syntactic analysis phase in greater detail. Input into a TXL [Cordy05] program is parsed according to the input grammar specified in the TXL program before the transformation rules are applied to it. The lexical and syntactic analysis phase of the TXL-based PT Pascal compiler/interpreter is implemented by taking advantage of this parsing phase. By specifying the PT Pascal syntax as a TXL input grammar lexical and syntactic errors in a PT Pascal source program can be efficiently detected. The PT Pascal source program is used as input into the lexical and syntactic analysis phase. The output from this phase is an identical source program since this phase only detects lexical and syntactic errors and does not apply any transformations to the source program.

This chapter goes on to demonstrate how the lexical components of a language such as keywords and identifiers can be specified in a TXL grammar in order to perform lexical analysis. It also discusses how the syntactic structures in a language can be specified in order to parse an input source program. Examples of the S/SL implementation of a scanner and parser for PT Pascal are used to illustrate the differences between traditional implementations of the lexical and syntactic analysers and the TXL implementation.

4.1 Lexical Analysis

Lexical analysis involves breaking up the characters in the source program into language tokens. This includes identifying keywords, compound tokens, comments and identifiers in the source program.

Keywords in the input language are specified in the TXL input grammar using the keys statements. The keys statement in the TXL input grammar for PT Pascal is shown in Listing 4.1 below as an example.
Listing 4.1: Keys statement in PT Pascal grammar

In the S/SL implementation of the PT Pascal compiler lexical analysis is performed by the Scanner and Screener routines. The Scanner breaks up the text of the source program into PT Pascal language tokens. It makes no distinction between identifiers and keywords and emits identifier tokens in both cases. Keywords are added to the identifier table during initialisation. The Screener then replaces identifier tokens with their keyword values by looking up the identifier table in which keywords are entered on initialization as can be seen in the code in Listing 4.2 taken from the original implementation of the PT Pascal compiler [Ross80].

Listing 4.2: Screening of identifiers in PT Pascal compiler

```
keys
'
end keys

{ Lookup the last accepted ident in the ident table using a hash function. If the ident is not in the table enter it. }
IdentLookup;

if idTokenValue[hashValue] <= 0 then

{ Replace the identifier token by its keyword value }
parseInputToken := idTokenValue[hashValue] +
    lastKeywordToken
else

{ Set the value of a compound user defined identifier token }
parseTokenValue := idTokenValue[hashValue];
```

Since the TXL engine handles the creation and management of the identifier table and the identification of keywords, these implementation details are hidden from the user when creating a lexical analyser using TXL.

TXL parses each special character in the input as a separate terminal symbol. If a sequence of special characters is to be treated as a single terminal symbol a compound statement containing
the desired sequence can be added to the TXL input grammar. PT Pascal compound tokens that are specified in the TXL input grammar are shown below in Listing 4.3.

```
compounds
  <> <= >= :=
end compounds
```

Listing 4.3: Compounds statement in PT Pascal grammar

Compound tokens are recognized in the Scanner by the order in which they appear in the input stream. Using the ‘:=’ symbol as an example, if a ‘:’ character is accepted, it is consumed but no language token is emitted until the next character in the input stream is examined. If the next character is a ‘=’ it is consumed and the language token pColonEquals is emitted i.e. it is recognized as the compound token ‘:=’. On the other hand, if the next character is not a ‘=’ it is accepted but not consumed and a pColon token is emitted. An S/SL code snippet that accepts a ‘:=’ symbol can be seen in Listing 4.4 taken from the original implementation of the PT Pascal compiler [Ross80].

```
| ':':
  |
  | '="':
    .pColonEquals
  | '*':
    .pColon
|
```

Listing 4.4: Scanning a compound token in S/SL

The comments statement in TXL allows a user to specify the style of comments in the input language. TXL ignores comments in the source program by default. The syntax of comments in PT Pascal is described in the comments statement shown below in Listing 4.5.

```
comments
  { }
  (* *)
end comments
```

Listing 4.5: Comments statement in PT Pascal grammar
In the S/SL implementation of the PT Pascal compiler comments are identified by the matching
the comment delimiter and then consuming all the characters in between without emitting any
tokens. A listing of the code that discards the contents of a comment is shown in Listing 4.6 taken
from the original implementation of the PT Pascal compiler [Ross80].

```
| '{':  % Discard the contents of a comment
  [  
    | '}'':
    >  
    | lNewLine:  
    .pNewLine
    | lEndFile:  % comment must be closed before EOF  
    #eCommentEOF
    .pEndFile
    >
    | '*': % comment may contain any other character  
    ?
  ]};
```

Listing 4.6: Scanning a comment in S/SL

TXL has a set of predefined nonterminals that match basic input tokens such as identifiers and
integers. The tokens statement can be used to define more input token types or modify the
predefined input types. Identifiers in PT Pascal can be any sequence of letters and digits starting
with a letter. Identifiers as specified by the TXL predefined type [id] may however also contain
underscores. The input type [id] is modified using the tokens statement in the TXL input grammar
so that only identifiers that are legal in PT Pascal are matched by the nonterminal [id]. Also, the
character `\` is used as the escape character in TXL but PT Pascal uses the character `. The
predefined nonterminal [charlit] must therefore be modified using the tokens statement to
accommodate this change as shown in Listing 4.7.

```
tokens
  id  "\\a[\\d\\a]*"
  charlit  "'\{''\}#'*''
end tokens
```

Listing 4.7: Tokens statement in PT Pascal grammar
Any unquoted sequence of letters and digits starting with a letter is identified as an identifier by the Scanner and a pIdentifier token is emitted. Each character in the sequence is added to a buffer when it is consumed using the S/SL semantic mechanism oBufferSave. When the end of the sequence is reached the contents of the buffer are also emitted along with the pIdentifier token. Any quoted sequence of characters is emitted with a pLiteral token. The S/SL code that scans an identifier is shown in Listing 4.8 taken from the original implementation of the PT Pascal compiler [Ross80].

```
  | lLetter:
    oBufferSave
  {[
      | lLetter, lDigit:
        oBufferSave
      | *:
        .pIdentifier
    ]}
```

Listing 4.8: Scanning an identifier in S/SL

### 4.2 Syntactic Analysis

A TXL grammar consists of a set of define statements that describe the forms of the nonterminals in the grammar. The entire input has to be of the nonterminal type [program] by convention. The TXL grammar must therefore contain a definition for the nonterminal [program]. In the input grammar for PT Pascal the structure of the entire program is described in the definition of the [program] nonterminal shown in Listing 4.9.

```
define program
  'program [name] ([list fileVariable+] ); [NL][IN]
  [block].
end define
```

Listing 4.9: Program definition in PT Pascal grammar

The checking of the syntactic structure of the source program is done by the Parser routine in the S/SL implementation of the PT Pascal compiler. The Parser consumes tokens output by the Scanner/Screener, emits tokens that identify statements, disambiguate operators and converts
expressions to postfix form. The tasks carried out by the Parser are specified in S/SL. The order in which the input tokens are expected is laid out in the S/SL program and tokens are emitted when appropriate. If a token that is expected is not received from the input stream a syntax error is emitted. The S/SL specification of the syntax of a program in PT Pascal is shown below in Listing 4.10 taken from the original implementation of the PT Pascal compiler [Ross80].

As seen in the example in Listing 4.9 a nonterminal is defined as a sequence of terminals and nonterminals. Nonterminals are enclosed in square brackets. A nonterminal can have more than one form. The various forms of the nonterminal are listed as ordered alternatives in the define statement. Constants in PT Pascal are an example of this since a constant can either be an integer, a literal or a constant identifier. The Boolean values ‘true’ and ‘false’ are predefined constant identifiers. The definition of the [constantValue] nonterminal is shown in Listing 4.11.
Similarly, different syntactic forms of the same nonterminal are specified using the choice action in S/SL. The specification of constants in the S/SL implementation of the PT Pascal compiler is shown below in Listing 4.12 as an example, taken from the original implementation of the PT Pascal compiler [Ross80].

```
define constantValue
  [charlit]
  | [opt '-'][integernumber]
  | [booleanValue]
  | [name]
end define

define booleanValue
  'true
  | 'false
end define

ConstantValue : 
  '='
  |
  pInteger:
    .sInteger
  | pIdentifier:
    .sIdentifier
  | '-':
    @UnsignedIntegerConstant
    .sNegate
  | '+':
    @UnsignedIntegerConstant
  | pLiteral:
    .sLiteral
  |
  ';' ;

UnsignedIntegerConstant : 
  |
  pIdentifier:
    .sIdentifier
  | pInteger:
    .sInteger
};
```

Listing 4.11: Definition of program parameter in PT Pascal grammar

Listing 4.12: S/SL specification of PT Pascal constants
TXL has nonterminal modifiers that can be used to indicate that items are optional or to match lists or sequences of an item. These modifiers are very useful in describing the syntax of a language. The modifier ‘opt’ is used to indicate that the item to be matched is optional. The variable references have subscripts only if the variables are of type array. The syntax for the variable references can be easily defined using the ‘opt’ keyword as shown in the example in Listing 4.13.

```
define optSubscriptedName
    [name][opt subscript]
end define

define subscript
    '[[expression]]'
end define
```

Listing 4.13: Example of ‘opt’ modifier use in PT Pascal grammar

In the S/SL implementation of the PT Pascal compiler an otherwise clause in the choice action, with no actions in it, is used when handling an optional item. The handling of variable references with optional subscripts is shown in Listing 4.14 below as an example, taken from the original implementation of the PT Pascal compiler [Ross80].

```
| pIdentifier:
    .sIdentifier
    [ |
        | '[:
            .sSubscript
            @Expression ']
            .sExpnEnd
        |
    *:
;
```

Listing 4.14: Scanning optional subscript in the S/SL

Sequences of zero or more items of a type can be matched using the ‘repeat’ modifier. A term is an expression in PT Pascal is a factor followed by an optional sequence of binary operator and factor pairs. This can be specified in the input grammar with the help of the ‘repeat’ keyword as seen in Listing 4.15.
Sequences of zero or more items can be matched using the cycle action in S/SL. The specification of a term, which may contain zero or more binary operator factor pairs, in the S/SL implementation of the PT Pascal compiler is shown in Listing 4.16 as an example, taken from the original implementation of the PT Pascal compiler [Ross80].

```
define term
    [factor] [repeat binaryOperatorFactor]
end define

define binaryOperatorFactor
    [factorOperator] [factor]
end define
```

Listing 4.15: Example of 'repeat' modifier use in PT Pascal grammar

If the sequence to be matched must contain at least one item the ‘repeat’ modifier followed by a ‘+’ symbol after the nonterminal can be used. In PT Pascal, the keyword ‘var’ must be followed by at least one variable declaration. This is specified using the ‘+’ symbol after the nonterminal modified by the ‘repeat’ keyword as can be seen in the example in Listing 4.17.

```
Term :
    @Factor
    {
        | '*':
            @Factor .sMultiply
        | 'div':
            @Factor .sDivide
        | 'mod':
            @Factor .sModulus
        | 'and':
            .sInfixAnd @Factor .sAnd
        | ':':
            >
    }
```

Listing 4.16: S/SL specification of a term in the PT Pascal compiler
In the S/SL implementation of the PT Pascal compiler a sequence of one or more items is specified by using match actions to specify one occurrence of the item and then using a cycle action to accept more occurrences of the same item. This is demonstrated in the specification of a variable declaration that is shown in Listing 4.18 taken from the original implementation of the PT Pascal compiler [Ross80].

Comma separated list of items can be specified using the ‘list’ modifier. Again, as with the ‘repeat’ modifier, the ‘list’ modifier matches zero or more items. By adding a ‘+’ symbol after the modified nonterminal it can be forced to match at least one item. The labels in a case statement in PT Pascal are a comma separated list and must contain at least one label as seen in Listing 4.19.

Define variables

```
define variables
    'var' [NL][IN]
    [repeat variableDeclaration+][EX]
end define
```

Define variableDeclaration

```
define variableDeclaration
    [name] ': ' [typeBody] ; [NL]
end define
```

Listing 4.17: Example of sequence with one or more items in the PT Pascal grammar

Listing 4.18: S/SL specification of variable declarations in PT Pascal compiler

Listing 4.19: Example of 'list' modifier use in PT Pascal grammar
Comma separated lists are specified in S/SL using the cycle action as they are used for specifying sequences and the commas are explicitly specified using a match action. The specification of case labels in the S/SL implementation of the PT Pascal compiler is shown in Listing 4.20 as an example, taken from the original implementation of the PT Pascal compiler [Ross80].

Listing 4.20: S/SL specification of case labels in PT Pascal compiler

After TXL transforms are run on the input, the transformed trees are unparsed. Formatting of the output from the TXL transforms can be done by adding TXL’s built-in formatting nonterminals to the input grammar. A new line can be started using the nonterminal [NL]. Indents and exdents can be added using the nonterminals [IN] and [EX] respectively. Since the output from an interpreter is not in the form of source code, it is not essential to include formatting in the PT Pascal input grammar. However, properly formatted output is useful when debugging the interpreter, since the output from the intermediate phases of the interpreter will probably be examined then. The formatting of a repeat until statement in PT Pascal is shown in Listing 4.21 as an example.

Listing 4.21: Example of formatting nonterminal use in PT Pascal input grammar
Some of the advantages of implementing a lexical and syntactic analyser in TXL, as demonstrated in this chapter, are as follows. Since a lexical and syntactic analyser can be implemented in TXL by specifying a grammar for the language in question the details of implementation of a scanner/parser are hidden from the user. TXL grammars have unrestricted forms and permit ambiguities which make it easy to rapidly create a grammar for a programming language and to modify the grammar to include new language features. Keywords can be easily distinguished from other identifiers by using the keywords statement. Comment styles can also be specified with ease using the comments statement and new lexical tokens can be specified using the tokens statement. On the other hand, hiding the details of implementation in this phase makes it more difficult to demonstrate how a parser handles tasks such as backtracking and error recovery.

In this chapter the implementation of the lexical and syntactic analysis phase by specifying the grammar for PT Pascal in TXL was discussed in detail. The next chapter describes the implementation of the semantic analysis phase. The semantic analysis phase is only run if no errors are found in this phase.
Chapter 5
Semantic Analysis

The lexical and syntactic analysis phase is implemented by specifying a TXL [Cordy05] grammar for PT Pascal [Ross80] as described in the previous chapter. The semantic analysis phase is run after the lexical and syntactic analysis phase and is discussed in this chapter. In the semantic analysis phase of the TXL-based PT Pascal compiler/interpreter information about the declared symbol in the source program is compiled and used to perform scope analysis and symbol and type checking. The variable references in the source program are also annotated with the kind and type information to be used in later phases.

This chapter discusses how a variety of semantic analysis tasks can be accomplished using a source transformation system and also illustrates various TXL programming paradigms. It demonstrates how a subgrammar and global variable can be used to implement a symbol table and discusses rules that can manipulate the table. It also shows how scopes can be incorporated into the symbol table and how the information stored in this table is used to perform symbol and type checking among other tasks.

Where appropriate the TXL implementation of the semantic analysis phase is contrasted against the traditional implementation of the PT Pascal compiler. This implementation of the semantic analysis phase differs in many respects from the previous implementation as would be expected. While the semantic checks performed by the TXL implementation of the Semantic Analyser are the same as the ones performed in the traditional implementation, the source program is not converted in a t-code instruction sequence. Instead the output from this phase is a typed intermediate source.

5.1 Symbol Table

During semantic analysis the translator requires information about the names declared in the program in order to carry out the various analysis tasks. A symbol table is a data structure in which this information about the names that appear in the source program is stored. Entries in a
symbol table consist of the name of the symbol and other information associated with the name [Aho77].

Data structures can be created in TXL by creating a grammar definition for the structure. In the TXL implementation of the semantic analyser the grammar definition of the symbol table and the rules that perform basic operations on it such as adding or finding a symbol are located in the symbol table module.

A nonterminal [frame] is identified by an item of type [frameNumber] and contains a sequence of items of the type [symbol]. The grammar definition of the nonterminal [frame] is shown below in Listing 5.1.

```plaintext
define frame
    [frameNumber] [NL]
    [repeat symbol][NL]
end define

define frameNumber
    'frame [number]
end define
```

Listing 5.1: Grammar definition of the nonterminal [frame]

The nonterminal [symbol] is defined as shown in Listing 2. It consists of an item of the nonterminal type [symbolName] and an item of type [symbolInfo]. The nonterminal type [symbolInfo] is intended to contain the information about a declared symbol in the program source. The grammar definitions for [symbolName] and [symbolInfo] are also included in Listing 5.2.
Each entry into the symbol table is an item of the type `symbol`. An item of the type `frame` is a sequence of symbol entries for the symbols in a scope of the source program. The symbol table is a sequence of frames and is implemented as a global variable of the type `repeat frame`. An example of the symbol table created for the symbols in a source program is shown in Figure 5.1.
In each symbol table entry the name of the symbol is stored as an item of the type [symbolName]. The symbol is identified as being a constant, type, variable, procedure or variable parameter and this information is stored as an item of type [symbolKind]. The [value] nonterminal is used to store the values of constants or the file descriptors for file variables. The type of the symbol is stored in the nonterminal [typeKind]. Symbol entries can have an optional item of the type [bounds] that is used to store the lower and upper bounds of a subrange or an array. The type of the component of an array or file symbol is stored as an optional item of type [componentType]. If the component is of type subrange another optional item of type [componentBounds] is used to store the bounds of the component subrange.

The optional item of type [numberParameters] is present in entries for procedure symbols and stores, as suggested by the type, the number of parameters in the procedure declaration. A frame
that contains the symbol table entries for the parameters of the procedure can be added to the symbol table entry for the procedure as an item of the type [parameterInfo]. The lexical level order number address that is generated for a variable identifier when it is entered into the symbol table is added to its symbol table entry as an item of type [llon]. An item of type [fileMode] is added to the symbol table entries for file variables and it used to indicate whether a file is open for reading or writing. Examples of the symbol table entries for different kinds of symbols can be seen in the sample symbol table in Figure 5.1.

In this implementation of the Semantic Analyser both symbol and type information is stored in the same table and as mentioned before the symbol table is implemented as a global variable so that it is easily accessible to all the rules in the Semantic Analyser and any changes made to it are reflected in all the rules.

Listing 5.3: Symbol table in the TXL-based PT Pascal compiler/interpreter

In the S/SL implementation of the PT Pascal compiler the symbol information and type information are stored in two separate tables and each entry in the symbol table is linked to an entry in the type table where the symbol’s type information is stored. The tables are implemented as a set of arrays as shown in Listing 5.4 taken from the original implementation of the PT Pascal compiler [Ross80].

Listing 5.4: Symbol table in the PT Pascal compiler

The body of each declaration in the program source is passed as an argument to a sequence of TXL functions that extract and return the attributes of the symbol which are then grouped together into a symbol table entry for that symbol with the help of a TXL constructor. The symbol entry is
then added to the symbol table. Functions that perform operations on the symbol table, such as
adding a symbol table entry, are part of the symbol table module. Listing 5.5 shows the TXL rule
that creates symbol table entries for constants and adds them to the symbol table.

Listing 5.5: TXL rule that creates and adds symbol table entries for constants

```
rule createConstSymbol
  replace [repeat constantDefinition]
    Id [id] ' = TypeBod [typeBody] ;
  RestConstDefs [repeat constantDefinition]

  % find the type kind of the constant
  construct TypeKind [typeKind]
     'Identifier
  construct NewTypeKind [typeKind]
     TypeKind [getIntType TypeBod]
     [getCharType TypeBod]
     [getBooleanType TypeBod]
     [getTypeFromConstantId TypeBod]

  % find the value of the constant
  construct Value [value]
     TypeBod [getNegativeIntegerValue]
     [getValueFromConstantId]

  % create a symbol for the constant
  construct NewSymbol [symbol]
     Id 'Constant Value NewTypeKind

  % add the symbol to the table
  import SymbolTable [repeat frame]
  export SymbolTable
     SymbolTable [previouslyDeclaredError Id]
     % emit an error message if constant has the
     % same name as a program parameter since
     % program parameters can only be file
     % variables
     [externalNotFileError Id NewSymbol]
     [addSymbol NewSymbol]

  by
     RestConstDefs
end rule
```

Before the new symbol table entry is added to the symbol table a check is performed to see if a
symbol with the same name is already present in the first frame of the table. If that is found to be
the case an appropriate error message is emitted and the older entry is removed from the symbol
table. The function that performs the check and emits the error message is shown in Listing 5.6 followed by an example of the error message that is emitted in Figure 5.2.

```txl
function previouslyDeclaredError SymName [id]
    replace [repeat frame]
        Tbl [repeat frame]
        % only look for repeated id in uppermost frame since that is
        % the frame that represents the current scope of the program
        deconstruct Tbl
            FrNum [frameNumber] SymList [repeat symbol]
            RestFr [repeat frame]
        deconstruct * SymList
            SymName Sinfo[symbolInfo]
            RestofList [repeat symbol]
        deconstruct not * [symbolKind] Sinfo
            'External
        construct ErrorMsg [stringlit]
            "Identifier declared twice: 
        construct FullMsg [stringlit]
            ErrorMsg [quote SymName]
                [print]
        import NumErrors [integernumber]
        export NumErrors
            NumErrors [+ 1]
    by
        Tbl [removeSymbol SymName] % if identifier has been
            % declared twice
            % remove previous symbol
end function
```

Listing 5.6: TXL function that emits an error message if a symbol is doubly declared
Figure 5.2: Example of error message emitted when a variable is doubly declared

The symbol table module also contains rules that perform functions such as adding an entry to the symbol table or retrieving an entry given the name of the symbol. A new symbol entry is added to the symbol table by replacing the sequence of symbol entries in the uppermost frame of the table by a new sequence that consists of the new symbol entry followed by the old sequence of symbol entries. The TXL function that adds a symbol to the symbol table is shown in Listing 5.7.

Listing 5.7: TXL function that adds a new symbol entry to the symbol table

```txl
% adds a symbol to the top of the uppermost frame on the symbol % table
function addSymbol Newsymbol [symbol]
    replace [repeat frame]
        FrNum [frameNumber] SymList [repeat symbol]
        RestFr [repeat frame]
    construct NewFr [frame]
        FrNum Newsymbol SymList
    by
        NewFr RestFr
end function
```

Semantic Analyser

Identifier declared twice: a
Number of errors found: 1
In the S/SL implementation of the PT Pascal compiler symbol information is collected as declarations are processed and stored on symbol and type stacks. This information is then entered into the symbol and type tables when a call to the oSymbolTableEnter mechanism is made. The implementation of the oSymbolTblEnter operation is shown in Listing 5.8 taken from the original implementation of the PT Pascal compiler [Ross80].

**Listing 5.8: S/SL semantic operation that enters a symbol into the symbol table**

```pascal
oSymbolTblEnter:
{ Create a symbol table entry with the attributes of the top symbol stack entry and link it to the top symbol stack entry. }
if symbolTblTop < symbolTblSize then
begin
    symbolTblTop := symbolTblTop + 1;
    symbolTblKind[symbolTblTop] := symbolStkKind[symbolStkTop];
    symbolTblValue[symbolTblTop] := symbolStkValue[symbolStkTop];
    symbolTblTypeTblLink[symbolTblTop] := symbolStkTypeTblLink[symbolStkTop];
    symbolStkSymbolTblRef[symbolStkTop] := symbolTblTop;
    { Update identifier table links }
    link := symbolStkIdentTblRef[symbolStkTop];
    if link > 0 then
        { This is a normal identifier, not a dummy identifier generated by the parser's syntax error recovery procedure. }
        begin
            symbolTblIdentLink[symbolTblTop] := identSymbolTblRef[link];
            identSymbolTblRef[link] := symbolTblTop;
        end
end
```

In the TXL implementation a symbol can be looked up in the symbol table by performing a searching deconstruct on the table using the name of the symbol to identify the relevant entry. The getSymbolFromTable function returns a copy of the entry associated with the given symbol name. If there is no entry found for the symbol name in question the searching deconstructor fails and nothing is returned. The TXL function that returns a symbol table entry given a symbol name as a parameter is shown in Listing 5.9.
Listing 5.9: TXL function for looking up a symbol in the symbol table

In the S/SL implementation of the PT Pascal compiler the identifier is looked up in the symbol table and if an entry is found in the symbol table it is pushed onto the symbol stack as shown in Listing 5.10 taken from the original implementation of the PT Pascal compiler [Ross80].

```pascal
deconstruct * [symbol] SymbolTable
    SymName SymInfo [symbolInfo]
```

Listing 5.10: Symbol table lookup in the PT Pascal compiler

5.2 Implementation of Scopes

Scopes are implemented with the help of the frames in the symbol table. The symbol table is treated like a stack with an empty frame being pushed onto it when a new scope is entered and the
same frame being popped off when the scope is exited. All the symbol table entries for the symbols that are local to the scope are stored in this new frame. Symbols that are local to a scope are not visible outside the scope since the frame that contains their symbol table entries is removed from the table when the scope is exited. The example in Figure 5.3 illustrates this.

When pushing a new frame is onto the symbol table an empty frame is first created using a TXL constructor. A constructor in TXL is used to bind a subtree to a new variable. After the empty frame is created the sequence of frames in the symbol table is replaced with a new sequence of frames starting with the empty frame followed by the original sequence of frames.

```
% puts a frame with no symbols in it on the top of the symbol table
function pushEmptyFrame
    replace [repeat frame]
        Frames [repeat frame]
        deconstruct * [frameNumber] Frames
        'frame FrNum [number]
        construct NewFr [frame]
        'frame FrNum [+ 1]
    by
        NewFr Frames
end function
```

Listing 5.11: TXL function that pushes an empty frame onto the symbol table

Symbols are looked up in the symbol table using searching deconstructs. A searching deconstruct always returns the leftmost shallowest subtree that matches the specified pattern. As a consequence, when a symbol is looked up the entry for the locally declared symbol is returned even if a symbol of the same name has been declared in an outer scope since the entries for the symbols declared in the current scope are added to the first frame in the sequence of frames in the symbol table. This ensures that the local definition of a symbol overrides any definition of the symbol in an outer scope.
program myprog (output);
  var
    var1 : char;
  procedure proc (var2 : char);
    var
      var3 : char;
    begin
      begin
      end;
  begin
  end.

Symbol Table – before procedure

frame 0
  var1 Variable 0 char % (0, 0) var1
  output Variable 2 file text

Symbol Table – inside procedure

frame 1
  var3 Variable 0 char % (1, 1) var3
  var2 Variable 0 char % (1, 0) var2

frame 0
  var1 Variable 0 char % (0, 0) var1
  output Variable 2 file text

Symbol Table – after procedure

frame 0
  proc Procedure 0 null 1
  frame -1
    var2_parm Variable 0 char
  var1 Variable 0 char % (0, 0) var1
  output Variable 2 file text

Figure 5.3: Example of scopes in the TXL implementation of the symbol table

After exiting the scope of a procedure it is necessary to retain the symbol table entries created for
the parameters of the procedure since parameter type information is required when checking the
compatibility of arguments in a call statement. A copy of the local frame is obtained after entries
for the parameters are added to it by calling the TXL function shown in Listing 5.12. These
symbols are renamed so they cannot be accessed outside the scope of the procedure by the rule shown in Listing 5.13. This frame is then added as an attribute to the procedure’s symbol table entry.

Listing 5.12: TXL function that returns a frame with parameter symbol entries

function getParameterFrame
  replace [opt frame]
    _ [opt frame]
  import SymbolTable [repeat frame]
deconstruct SymbolTable
  'frame _ [number] SymList [repeat symbol]
    RestFr [repeat frame]
  construct Zero [number]
    0
  by
    'frame Zero [- 1] SymList [__removeLlonFromParams]
      [__addUnderScoreToParamName]
end function

Listing 5.13: TXL rule that renames parameter symbols

rule __addUnderScoreToParamName
  replace $ [symbol]
    SymName [id] SymInfo [symbolInfo]
  by
    SymName [__ 'parm] SymInfo
end rule

A frame is popped off the symbol table by replacing the sequence of frames that make up the symbol table with a sequence starting from the second frame.

Listing 5.14: TXL function that pops a frame off the symbol table

% removes uppermost frame from the symbol table. Used when exiting % the scope of a procedure

function popFrame
  replace [repeat frame]
    Fr [frame] RestFr [repeat frame]
  by
    RestFr
end function
In the S/SL implementation of the PT Pascal compiler symbols declared within a scope are pushed onto the symbol table when the scope is entered and are then popped off the symbol table when the scope is exited. Parameter symbols are left in the symbol table but are disassociated from their names. The active scopes are kept track of using a lexical level stack. The stack contains pointers to the base entries of all the active scopes in the symbol table. When a new scope is entered its base entry pointer is pushed onto the lexical level stack by calling the oSymbolTablePushScope operation. The oSymbolTablePushScope operation is shown in Listing 5.15 taken from the original implementation of the PT Pascal compiler [Ross80].

```
oSymbolTblPushScope:
   { Push the lexic level stack with pointers to the symbol and type stack tops. }
   if lexicLevelStackTop < lexicLevelStackSize then begin
      lexicLevelStackTop := lexicLevelStackTop + 1;
      symbolTblDisplay[lexicLevelStackTop] := symbolTblTop;
      typeTblDisplay[lexicLevelStackTop] := typeTblTop;
   end
   else
      Error(eLexicLevelStackOvfl);
```

Listing 5.15: S/SL semantic operation for entering a scope in the PT Pascal compiler

When a scope is exited the base entry pointer for the scope is popped off the lexical level stack, local symbol entries are removed from the symbol table and parameter entries are kept. All the types defined in the scope being exited are also popped off the type table. The oSymbolTblPopScope operation that is called by the S/SL program when a scope is existed is shown in Listing 5.16 taken from the original implementation of the PT Pascal compiler [Ross80].
Symbol checking is done to ensure that references to symbols in the source program are legal in
the language. In the TXL-based PT Pascal compiler/interpreter symbol checking is done by
examining the [symbolKind] attribute in the symbol table entry of the symbol in question. Once
the symbol table entry for the symbol has been returned a TXL searching function, that matches

Listing 5.16: S/SL semantic operation for exiting a scope in the PT Pascal compiler

5.3 Symbol and Type Checking

Symbol checking is done to ensure that references to symbols in the source program are legal in
the language. In the TXL-based PT Pascal compiler/interpreter symbol checking is done by
examining the [symbolKind] attribute in the symbol table entry of the symbol in question. Once
the symbol table entry for the symbol has been returned a TXL searching function, that matches
the `[symbolKind]` attribute in the entry, is applied to it. An example of the error message generated when an error is detected during symbol checking is shown if Figure 5.4.

```plaintext
program myprog (output);
type
t = char;
begin
t := 'a'
end.
```

![Figure 5.4: Example of symbol checking done by the Semantic Analyser](image)

A searching function in TXL applies its replacement to the first subtree within the scope that matches its pattern. In case of functions that match a subtree but do not transform it, the match keyword can be used in place of the replace keyword and the by clause is eliminated. Conditions are expressed in TXL with the help of where clauses. Rules are applied to the variable in the where clause which succeeds if the variable matches the pattern in at least one rule. If a where clause fails the rule does not proceed. A where clause can be modified using the keyword not. A where not clause succeeds if the variable does not match the pattern in any of the rules applied. In the function in Listing 5.17 an error message is emitted if the symbol is not a variable or a variable parameter.
In order to perform type checks it is necessary to first evaluate the type of an expression. In the TXL-based PT Pascal compiler/interpreter the expression checker module contains the rules that simulate the evaluation of the expression in order to obtain its type. While simulating the evaluation of the expression the types obtained are stored as items of type [expressionType] which in turn consists of an item of type [operandType] and an item of type [resultType]. The grammar definitions for these nonterminals are also in the expression checker module.

```txl
function variableRequiredError Stmt [statement]
    match * [symbolKind]
        SymKind [id]
    where not
        SymKind [= 'Variable']
            [= 'VarParameter']
    construct ErrorMsg [stringlit]
        "Variable required: "
    construct NewStmt [statement]
        Stmt [removeStatements]
    construct FullMsg [stringlit]
        ErrorMsg [quote NewStmt]
        [print]
    import NumErrors [integerNumber]
    export NumErrors
        NumErrors [+ 1]
end function
```

Listing 5.17: TXL function that emits an error message if the symbol is not a variable

While the evaluation of the expression is simulated error messages are emitted if any type mismatches are encountered within the expression and symbol checking is performed when
symbols are encountered in the expression. Error messages are also emitted if an array variable is not subscripted or a non-array variable is subscripted. An example of an error message emitted during expression checking is shown in Figure 5.5.

The type of a factor is obtained by retrieving the [typeKind] attribute from the symbol table entry in the case of a variable reference. If the factor is a literal value the type is inferred from the factor itself. The results from calls to the PT Pascal predefined functions chr, ord, eof and eoln are also factors and their types are the same as the return type of the function. Listings 5.19, 5.20 and 5.21 show examples of the functions applied to retrieve the type from a factor in an expression.

Figure 5.5: Example of error message generated by the expression checker module
Listing 5.19: TXL function that returns the type of a variable referenced in an expression

```txl
definition function getTypeFromId Factor [factor] Exp [expression]
  replace [opt operandType]
  OpType [opt operandType]
  deconstruct Factor
    Id [id]
  construct SymName [symbolName]
    Id
  construct Symbol [opt symbol]
    _ [getSymbolFromTable SymName]
    [identifierNotDeclaredError Id]
    [illegalKindInExpError Id Exp]
  deconstruct * [typeKind] Symbol
    SymType [typeKind]
    by
      SymType [switchSubrangeToInteger]
      [illegalOperandError Id Exp]
end function
```

Listing 5.20: TXL function that returns the type of an integer literal

```txl
definition function getIntegerTypeFromFactor Factor [factor]
  replace [opt operandType]
  OpType [opt operandType]
  deconstruct Factor
    Sign [opt '-'] Num [integer number]
    by
      'integer
end function
```

Listing 5.21: TXL function that returns the type of the result from a PT Pascal function

```txl
definition function getTypeFromEoln Factor [factor]
  replace [opt operandType]
  OpType [opt operandType]
  deconstruct Factor
    'eoln FileName [opt filename]
    construct NewFname [opt filename]
    FileName [checkFileNameForEoln Factor]
    by
      'Boolean
end function
```

A term consists of a factor followed by an optional sequence of operator factor pairs. The type of the factor is retrieved and stored as an operand type. An expression type is found for each of the
pairs with the type of the factor as the operand type and the type of the type of the result of the
operation as the result type. Since operators are unambiguous in PT Pascal the result type of an
operation can be inferred from the operator itself without having to consider the types of the
operands involved. The operand type and result type in the expression type are checked for
compatibility. Listing 5.22 shows the function that checks for operand operator type mismatches.

function operandOperatorTypeClashError ResType [opt resultType] [exp expression]
  replace [opt operandType]
    OpType [typeKind]
  deconstruct not ResType
    OpType
  construct ErrorMsg [stringlit]
    "Operand and operator types clash: 
  construct FullMsg [stringlit]
    ErrorMsg [quote Exp]
      [print]
  deconstruct ResType
    NewOpType [typeKind]
  import NumErrors [integer]
  export NumErrors
    NumErrors [+ 1]
  by
    NewOpType
end function

Listing 5.22: TXL function that emits error message for an operand operator type mismatch

As each new expression type is found its result type is checked against the result type of the
previous expression type found. A new expression type is then created by replacing the operand
type in the previous expression type with the result type from the new expression type. This
process is repeated until all the operator factor pairs have been checked. Then the operand type
from the first factor is checked against this expression type for type mismatches. The result type
from the expression type is then returned as the type of the entire term. An example of a term and
how its type is evaluated is illustrated in Figure 5.6 and a function that returns the type of a term
is shown in Listing 5.23.
Simple expressions in PT Pascal consist of a term followed by an optional sequence of operator term pairs. The type of a simple expression is evaluated in the same way that the type of a term is evaluated. In this case the type of the term is found first, which becomes the operand type and the result type of the operation is the result type in the expression type. An expression consists of either a simple expression or a simple expression followed by one operator simple expression pair where the operator is a comparison operator. The type of the expression is the same as the type of the simple expression in the case where it consists of just one simple expression.

If the expression consists of a simple expression followed by an operator simple expression pair it is a boolean expression. In this case the two simple expressions’ types are evaluated and an operand type mismatch error message is emitted if the two operands have different types since PT Pascal can only make comparisons between values with the same type.

In the S/SL implementation the type stack is used to evaluate the type of an expression. The expression is in postfix form. As each operand is accepted its type is pushed onto the type stack. When an operator is encountered the types of the operands are popped off the stack and checked for mismatches. The result type of the operation is then pushed onto the stack. When all the

Listing 5.23: TXL function that returns the type of a term in an expression

```txl
function getTermWithOperatorType Term [term] Exp [expression]
  replace [opt operandType]
    E [empty]
  deconstruct Term
    Factor [factor]
      OpFactorList [repeat binaryOperatorFactor]
    construct OpType [opt operandType]
      _ [getFactorType Factor Exp]
    construct ExpType [opt expressionType]
      _ [getOperatorFactorType Exp each OpFactorList ]
  deconstruct ExpType
    _ [opt operandType] ResType [opt resultType]
  construct TermType [opt operandType]
    OpType [operandOperatorTypeClashError ResType Exp]
  by
    TermType
end function
```
tokens in the expression have been accepted the type that remains on the type stack is the type of the expression.

```
1 * 2 * 3
```

Type checking is also carried out on statements. In assignment statements the type of the variable is found in the same way that the type of a variable reference in an expression is found, by retrieving the [typeKind] attribute from the variable’s symbol table entry. The type of the variable is then checked against the type of the expression being assigned to it. Error messages are emitted if the conditional and control expressions in if, while and repeat until statements are not boolean expressions. Case statements are checked to ensure that case expressions are integer expressions.

Figure 5.6: Example of evaluating the type of a term
In calls to the read and write procedures the types of the arguments are checked against the
component type of the file argument and appropriate type mismatch errors are emitted if
necessary. Some of the functions that check for type mismatches in calls to the read procedure are
shown in Listing 5.24.

```
function checkReadTextFileArgument Exp [expnOptColonExpn]
  ArgType [opt operandType] Stmt [statement]
  match [opt operandType]
    'text
    deconstruct ArgType
      TypeKind [typeKind]
    where not
    TypeKind [isCharType]
      [isIntegerType]
    construct RTy [id]
      'readProc  % need this in order to use the same read
      % variable error rule in this rule and in
      % rule checkNonTextFileArgument
    construct NewId [id]
      RTy [invalidReadVariableError Exp Stmt]
end function

function checkNonTextFileArgument Exp [expnOptColonExpn]
  ArgType [opt operandType] Stmt [statement] RorW [id]
  match [opt operandType]
    FileType [opt operandType]
    deconstruct not FileType
    E [empty]
  deconstruct not FileType
    'text
  deconstruct not FileType
    ArgType
  construct NewId [id]
    RorW [invalidReadVariableError Exp Stmt]
    [invalidWriteExpressionError Exp Stmt]
end function
```

Listing 5.24: TXL functions that type check the arguments in a call to the read procedure

### 5.4 Rewriting

During the semantic analysis phase some constructs in the source program are transformed in
order to make it easier to perform certain analysis tasks in this phase. They are also annotated
with information required by later phases. The rules that perform these rewriting tasks are
described in this section.

61
Calls to the read and write procedures in PT Pascal do not have a fixed form. They may have an optional file argument followed by any number of variable arguments in the case of a read call and write expressions in the case of a call to the write procedure. In the absence of a file argument in a read call it is assumed that the data is being read from the standard input stream which is associated with the file variable ‘input’. If a write call is made without a file argument it is assumed that the data is being written to the standard output stream identified by the file variable ‘output’. Rules are applied to read and write calls that transform a call with multiple arguments into a sequence of calls each having two arguments; a file argument and a single read variable or write expression depending on the procedure being called. Listing 5.25 shows the rules that are applied to a read call to transform it.

Listing 5.25: TXL rule that transforms read call into sequence of calls with two arguments

```
rule changeReadStmts
  replace $ [repeat statementOptSemicolon]
  'read '(ArgList [list expnOptColonExpn+])' Semi[opt ',;]
  RestStmts [repeat statementOptSemicolon]
construct Stmt [statement]
  'read '(' ArgList ')
export FileName [opt symbolName]
  construct NewArgList [list expnOptColonExpn]
    ArgList [getFileNameFromArgList]
import FileName
construct Fname [opt symbolName]
  FileName [makeStdinFileName] % if no file name is present
  % use the standard input file
construct ReadList [repeat statementOptSemicolon]
  [createReadCall Fname each NewArgList]
deconstruct not * [statementOptSemicolon] ReadList
  'read '(' ArgList ')') Semi
by
ReadList [, RestStmts]
end rule

function createReadCall Fname [opt symbolName] Exp [expnOptColonExpn]
  replace [repeat statementOptSemicolon]
    ReadList [repeat statementOptSemicolon]
deconstruct Fname
    FileName [name]
construct ReadStmt [statementOptSemicolon]
  'read (FileName, Exp) ;'
by
ReadList [, ReadStmt]
end function
```

Listing 5.25: TXL rule that transforms read call into sequence of calls with two arguments

62
The names of the read and write procedures are also changed to reflect the type of the file being accessed and the type of the argument. This is necessary since both integers and characters can be read from the same text file and reading and writing to an integer file is done differently from reading and writing to a text file. If an integer file is read from or written to the procedure names in the calls are transformed from read and write to read_int and write_int. If an integer is being read from a text file the procedure read is again transformed into read_int and if a character is being read from a text file it is transformed into read_char. If a text file is being written to the procedure name is not changed. File arguments are also added to calls to the readln and writeln procedures if they are not already present.

```
program myprog (input, output, intfile);
var
  i : integer;
  c : char;
  intfile : file of integer;
begin
  write('one : ', 1);
  writeln;
  read(i, c);
  rewrite(intfile);
  write(intfile, i);
  reset(intfile);
  read(intfile, i)
end.
```

Figure 5.7: Example of transformed read and write calls
References to type identifiers in the program source are replaced with their definitions and references to constant identifiers are replaced with constant values. This is done in the semantic analysis phase since the rules can leverage the information already stored in the symbol table. Variable identifiers are replaced with their lexical level order number addresses since these addresses are used by later phases during memory allocation. Variable references are also annotated with kind and type information since it is required by the code generation phase. A rule that replaces a constant identifier with its value is shown in Listing 5.26 and Listing 5.27 shows a rule that annotates variable references with kind and type information.

```
rule replaceIntegerConstantWithValue
  % variable or type in routine might have the same name as
  % a constant in the outer scope so must skip routines when
  % applying rule to outer scope
  skipping [routine]
  replace $ [integerConstant]
    Id [id]
  construct SymName [symbolName]
    Id
  construct Symbol [opt symbol]
    _ [getSymbolFromTable SymName]
  deconstruct * [symbolKind] Symbol
    'Constant
  deconstruct * [value] Symbol
    Val [integernumber]
  by
    Val
end rule
```

Listing 5.26: TXL rule that replaces constant identifiers with their constant values
Implementing the semantic analysis phase using a source transformation system like TXL serves as an example of how various semantic analysis tasks can be accomplished using this technology. Symbol tables can be implemented as global variables so that they are easily accessible to all the rules in the program. The structure of the symbol table is specified as a subgrammar in TXL. In this implementation the symbol table entry also contains the type information of the symbol. This avoids the need for a separate type table and the complexity added by needing to maintain links between the two. The ease with which sequences can be created and manipulated in TXL also simplifies the creation and maintenance of a symbol table.

Since sequences in TXL are so convenient to use the implementation of scopes can also be done without the help of a display stack which is used in traditional implementations. This is done by creating a data structure that is a sequence of frames to use as a symbol table. By modelling this sequence as a stack onto which a new frame is pushed when a scope is entered we can ensure that local definitions of symbols override global definitions since they will be encountered in the uppermost frame when the symbol table is searched. We can also ensure that local variables are not accessible outside their scope by popping the frame off the symbol table when a scope is exited.
The declarative style of TXL rules makes it easier to see the relationship between each language construct and the semantic checks that need to be carried out on it. It also makes the rules easy to comprehend which is very helpful if this phase is to be modified to accommodate extensions to the language being implemented. On the other hand, since TXL does not have conditional or loop statements these constructs have to be artificially created by using global variables as flags and counters which can sometimes be tedious. The strongly typed nature of TXL also results in multiple rules that perform the same tasks but take parameters of different types.

The implementation of the semantic analysis phase is described in this chapter. In the semantic analysis phase the symbol table is implemented as a global variable and its structure is specified as a separate subgrammar. Symbol and type checking are done by running various functions and rules on the statements in the source program and error messages are emitted when semantic errors are detected. The implementation of the execution phase in the TXL-based PT Pascal compiler/interpreter is described in the next chapter. This phase is run on the output from the semantic analysis phase and can therefore only be run if no errors are detected during semantic analysis.
Chapter 6
Execution Phase

The execution phase is run after the semantic analysis phase discussed in the previous chapter if there are no semantic errors detected. In the execution phase of the TXL [Cordy05] implementation of the PT Pascal [Ross80] interpreter, the typed intermediate code generated by the semantic analysis phase is directly executed to produce the output from the program. During execution runtime checks such as checking if the value assigned to a subrange variable is within bounds are also performed. Data structures are created to simulate the memory and file buffers necessary for execution.

This chapter describes the implementation of the execution phase. A table is used to simulate the memory required in this phase. The structure of this table and rules that manipulate it are discussed. This is followed by a description of the file buffers used and the rules that perform I/O operations. Examples and code listings are used to demonstrate how expressions are resolved and the PT Pascal statements are executed. The detection of runtime errors is also explained in detail.

6.1 Memory

The memory used in the execution phase is a data structure that is organised much like the symbol table used by the Semantic Analyser. The memory consists of a sequence of frames each of which contains a sequence of memory cells.

```txl
define memoryFrame
    frame [frameNumber] [NL][IN]
    [repeat memoryCell] [EX]
end define

define frameNumber
    [integerNumber]
end define
```

Listing 6.1: TXL grammar definition of a memory frame

A memory cell is created for each variable that is declared in the source program excepting file variables. File buffers are created for each file variable and are discussed in the next section. A
memory cell consists of a lexical level order number address and an item of type [memoryInfo]. An item of type [memoryInfo] consists of an item of type [value] which is used to store the value that is assigned to the variable when the program is executed. In addition to an item of type [value] a memory cell contains optional items of types [indexBounds], [subrangeBounds] and [varArgAddress]. The array index bounds of an array variable are added to its memory cell as an item of type [indexBounds]. In case of subrange variables or array variables with subrange components the subrange bounds are also added to the memory cell as items of type [subrangeBounds]. When a memory cell is created for a variable parameter the lexical level order number address of the variable argument associated with it in the call is added to its memory cell as an item of type [varArgAddress].

```
define memoryCell
  [llon] [memoryInfo]
end define

define memoryInfo
  [value] [opt indexBounds] [opt subrangeBounds]
  [opt varArgAddress] [NL]
end define
```

Listing 6.2: TXL grammar definition of a memory cell

The memory used in the execution phase is implemented as a global variable that contains a sequence of memory frames.

```
% initialise memory
export Memory [repeat memoryFrame]
  'frame 1
```

Listing 6.3: Implementation of memory in the execution phase

The module that contains the TXL grammar definition of the memory also contains rules that perform operations such as adding a new memory cell, returning a specific memory cell and changing the value in a memory cell among others. A new memory cell is added to the memory by replacing the sequence of memory cells in the first frame of the memory with a new sequence starting with the new memory cell followed by the old sequence of memory cells.
Listing 6.4: TXL function for adding a new memory cell to the memory

A single memory entry is created for each of the variables declared in the source program except in the case of array variables. A memory entry is created for each element in the array. Figure 6.1 shows an example of the memory entries created for the variables in a program.

```
program myprog (output);
var
  a : integer;
  b : 4..7;
  c : array [0..2] of char;
  d : array [0..2] of 2..6;
begin
end.
```

Figure 6.1: Example of the Memory in the execution phase

Each memory cell is identified by the lexical level order number address of the variable whose value is stored in the cell. When resolving a variable reference in the program source it has to be replaced by the variable’s value from the memory. The variable’s memory cell, which contains the variable’s value, can be retrieved from the memory by using a searching deconstructor that matches the specified lexical level order number address to locate the desired cell and then returning it.
After a memory cell is returned by the getMemoryEntry function it is necessary to check if the variable has been initiated before replacing the variable reference in the source program with the value from the memory cell. When memory cells are created for the variables declared in the source program they are initially given the value ‘no_value’. By applying a function that matches the pattern ‘no_value’ to the returned memory cell, an error message can be emitted if the variable is not initiated. In the execution phase any error results in the halting of the pass unlike the semantic analysis phase where the pass is completed in most cases even if errors are found. Figure 6.2 shows an example of the error message that is emitted if a variable that has not yet been initiated is referenced.
Figure 6.2: Example of error message produced when an uninitiated variable is referenced

When the value assigned to a variable changes in the course of executing the source program this change has to be reflected in the variable’s memory cell. This is done by finding the relevant memory cell with the help of a searching deconstructor as is done when retrieving a memory entry and then replacing the item of type [value] in the memory cell with the new value that is being assigned to the variable. Listing 6.7 shows the TXL function that replaces the value in a memory cell with a new value. If the variable being assigned to is a subrange variable an error message is emitted if the value is not within the subrange bounds. This is done by applying a searching function to the memory cell that matches the [subrangeBounds] item and then compares the value to the bounds matched by the function. This function is also shown in Listing 6.7. An example of the error message that is produced if the value being assigned to a subrange variable is outside the bounds of the subrange is shown in Figure 6.3.
function replaceValueInMemory Llon [llon] NewVal [factor]
    replace * [memoryCell]
        Llon MemInfo [memoryInfo]
    by
        Llon MemInfo [outOfSubrangeBoundsError NewVal Llon]
            [changeValue NewVal]
end function

function changeValue NewVal [factor]
    replace * [value]
        _ [value]
    by
        NewVal
end function

function outOfSubrangeBoundsError NewVal [factor] Llon [llon]
    match * [opt subrangeBounds]
        LowBound [integernumber] UppBound [integernumber]
    deconstruct NewVal
        Num [integernumber]
    where
        Num [< LowBound]
            [> UppBound]
    construct Msg [stringlit]
        "Value out of subrange bounds in assignment to: "
    deconstruct * [opt id] Llon
        Id [id]
    construct NewMsg [stringlit]
        Msg [quote Id]
    [put]
        [quit 1]
end function

Listing 6.7: TXL functions that replace the value in a memory cell and check subrange bounds
Figure 6.3: Example of error message produced when a value is outside subrange bounds

The lexical level order number addresses generated by the semantic analysis phase are not unique. It is therefore necessary to implement scopes in the memory. This is done with the help of the memory frames. The memory is implemented as a stack with memory frames being pushed onto it or popped off it. When a new scope is entered an empty memory frame is pushed onto the memory. The memory cells for the variables declared within this scope are added to this frame.

Listing 6.8: TXL function that pushes an empty frame onto the memory

When the scope is exited this frame is popped off. If the frame contains the memory cells of any variable parameters the values from these cells are copied into the memory cells of the associated variable arguments before the frame is discarded.
Listing 6.9: TXL functions that pop a frame off the memory and replace variable argument values

An example of the state of the memory before, during and after a procedure call is shown in Figure 6.4.
6.2 File Buffers

TXL has predefined functions that can be used to read or write files. The put and fput functions write a line to the standard output stream and a file respectively. A line can be read from the standard input stream or a file using the functions get and fget. In the PT Pascal program source the arguments in the calls to the write procedure need to be stored until a call to the writeln procedure is encountered. Similarly when a line is read from a file the data needs to be stored.
until it is all assigned to variables or a new line is read. The file buffers are data structures that can
be used to temporarily store this data.

A file buffer module contains the grammar definition of the structure and rules that perform
operations on the buffers. An item of type [fileBuffer] consists of items of the types [name],
[writeBuffer], [readBuffer] and [eofValue] along with an optional item of the type
[actualFileName]. Listing 6.10 shows the grammar definition of [fileBuffer].

```
define fileBuffer
    [name] [opt actualFileName] [writeBuffer] [readBuffer]
    [eofValue]
end define

define actualFileName
    [stringlit]
end define

define writeBuffer
    [repeat factor]
end define

define readBuffer
    [stringlit]
end define

define eofValue
    [integernumber]
end define
```

Listing 6.10: TXL grammar definition of [fileBuffer]

A file buffer is created for each of the file identifiers listed in the program header in the source.
The global variable FileBuffers which consists of a sequence of items of type [fileBuffer] is used
to hold all the file buffers that are created so they can be easily accessed. The file identifier itself
is stored as an item of type [name]. The user defined file identifiers are used to identify files
within the source program but are not the same as the names of the files in the operating system.
When writing to or reading from files the file names used in the operating system are required.
These names are provided as command line arguments and must be associated with their
respective file identifiers. This is done by storing the operating system file name as an item of
type [actualFileName]. Command line arguments are stored in a predefined TXL global variable
TXLargs which is a sequence of string literals where each string literal is one command line argument.

Data that is to be written to the file is stored as an item of type [writeBuffer] which is in turn a sequence of items of type [factor]. A line read from a file is stored as an item of type [readBuffer] which is a string literal. The [eofValue] is used to indicate that the end of file has been reached. It has a default value of ‘0’ which is changed to ‘1’ when the end of file is reached. Listing 6.11 shows a TXL function that creates a file buffer for a file identifier listed in the program header and adds it to FileBuffers.

Listing 6.11: TXL rule that creates file buffers for file identifiers

```txl
rule createOtherBuffers
replace [fileVariable]
  FileVar [name]
  deconstruct not FileVar
    'input
  deconstruct not FileVar
    'output

  % actual file name passed as a program parameter. program
  % parameters are stored in predefined global variable
  % TXLargs
  import TXLargs [repeat stringlit]

  % if no actual file name supplied emit an error and stop
  % execution
  construct NewArgs [repeat stringlit]
    TXLargs [fileNameMissingError FileVar]

  deconstruct TXLargs
    FileName [stringlit] RestArgs [repeat stringlit]
  export TXLargs
    RestArgs
  construct WriteBuff [writeBuffer]
  _
  construct ReadBuff [readBuffer]
    ""
  construct Buffer [fileBuffer]
    FileVar FileName WriteBuff ReadBuff 0
  import FileBuffers [repeat fileBuffer]
  export FileBuffers
    FileBuffers [. Buffer]
  by
end rule
```

Listing 6.11: TXL rule that creates file buffers for file identifiers
The file buffers module also contains functions that perform operations such as adding a value to the write buffer or reading a line from the file and storing it in the read buffer. A value is added to the write buffer by adding the value to the end of the sequence of items in the write buffer. When a line is read into the read buffer from a file the line is read one character at a time and is stored as a sequence of tokens. TXL ignores white spaces by default but in the case of text files white spaces also count as characters. The “-char” command line option is turned on before the file is read using the TXL built-in function pragma which is used to change TXL options during execution. The “-char” option forces TXL to accept white spaces in the input. Once the all the characters are read and stored as tokens the TXL built-in function quote is used to concatenate all the characters into a string literal which is then stored as the read buffer. Listing 6.12 shows the functions that read a line from a file as characters and construct a string literal from them.
Listing 6.12: TXL functions that read a line from a file as characters

When a character is to be read from the buffer TXL’s built-in substring function is used to separate the first character from the string literal in the read buffer. The remaining string is placed back in the read buffer and the first character is returned. If an integer is to be read from the read buffer all the leading white spaces in the string literal are skipped. Then all the non-blank characters until the next white space are collected and parsed into an integer literal. The remaining string is then placed back in the read buffer. The functions that are used to get an integer from the read buffer, skip leading white spaces and collect and parse non-blank characters into an integer literal are shown in Listings 6.13, 6.14 and 6.15 respectively.
function getIntFromReadBuffer FileId [name]
    replace * [fileBuffer]
    FileId FileName [opt actualFileName] WriteBuff
    [writeBuffer] Value [stringlit] Eof [integernumber]
    construct NewEof [integernumber]
    Eof [checkIfEOFReached FileName]
    deconstruct not Value
    ""
    export NumString [stringlit]
    ""
    construct NewValue [stringlit]
    Value [skipBlanks]
    [getNumberFromString]
    import NumString
    construct Temp [signedInteger]
    0
    construct NewTemp [signedInteger]
    Temp [parse NumString]
    deconstruct NewTemp
    Sign [opt '-'] Num [integernumber]
    construct Val [factor]
    Num
    construct NewVal [factor]
    Val [makeNegative Sign]
    export IntValue [factor]
    NewVal
    by
    FileId FileName WriteBuff NewValue Eof
end function

Listing 6.13: TXL function that gets an integer from the read buffer

function skipBlanks
    replace [stringlit]
    Value [stringlit]
    construct Character [stringlit]
    Value [: 1 1]
    deconstruct Character
    " "
    construct Length [integernumber]
    _ [# Value]
    construct NewValue [stringlit]
    Value [: 2 Length]
    by
    NewValue [skipBlanks]
end function

Listing 6.14: Recursive TXL function that skips leading white spaces
Listing 6.15: Recursive TXL function that collects non-blank characters

The default value of the [eofValue] attribute is ‘0’. This value is changed to ‘1’ when the line returned by the get or fget functions is empty. When a value is to be read from the read buffer the [eofValue] attribute of the file buffer is checked. If the value is found to be ‘1’ an end of file reached error message is emitted and execution is halted.
6.3 Statement Execution

In the interpretation phase file buffers are first created for each of the file variables listed in the program header in the source program. Memory cells are then created for each of the variables declared in the global scope of the source program which are added to the Memory. Once this is done each of the statements are executed in the order in which they appear in the source program.

6.3.1 Expression Evaluation

An expression is evaluated by applying rules that resolve both the variable references and the operators in the expression. There are separate rules that resolve subscripts, variables references and each operator. Operator precedence is preserved by calling the rules in decreasing order of the precedence of their operators. The operators are left associative since the rules traverse the expression tree from left to right. This implementation depends upon expressions being free of side effects in order to produce correct results, which is the case in PT Pascal since it does not have functions. Listing 6.16 shows the rule that is applied to expressions to evaluate them.

Figure 6.5: Example of error message emitted when end of file is reached

```pascal
program myprog (input, output, myf);
var
  % (0, 0) i : char;
  myf : text;
begin
  reset (myf);
  read_char (myf, % (0, 0) i);
  readln (myf);
  read_char (myf, % (0, 0) i);
end.
```

EOF reached. Cannot read from file : myfile.txt
A subscript is resolved by replacing the subscripted lexical level order number address with the address of the array element whose index corresponds to the value of the subscript expression. In order to do this the array index bounds are obtained from the [bounds] attribute in the memory entry of the variable and the index is obtained by evaluating the subscript expression. The index is checked against the bounds of the array before the subscript is resolved and an error message is emitted if the index is outside the bounds. The new address is then created. It has the same lexical level but a different order number that is calculated by adding the index to the old order number and then subtracting the lower bound from the result. Listing 6.17 shows the rule that resolves subscripts.

Listing 6.16: TXL rule that evaluates a PT Pascal expression

```txl
rule resolveExpression ExpForError [expression]
  replace [expression]
    Exp [expression]
  construct NewExp [expression]
    Exp [resolveSubscript]
      [resolveLlon]
      [resolveUnaryMinus]
      [resolveNot]
      [resolveMultiplication]
      [resolveDivision ExpForError]
      [resolveModulo]
      [resolveAnd]
      [resolveAddition]
      [resolveSubtraction]
      [resolveOr]
      [resolveEqual]
      [resolveNotEqual]
      [resolveLessThan]
      [resolveLessThanOrEqual]
      [resolveGreaterThan]
      [resolveGreaterThanOrEqual]
      [resolveChr]
      [resolveOrd]
      [resolveEof]
      [resolveEoln]
      [resolveBrackets]
    deconstruct not NewExp
    Exp
    by NewExp
end rule
```
Variable references are resolved by replacing the lexical level order number addresses in expressions with the values associated with them in the memory. The rule that resolves variable references is shown in Listing 6.18.

Listing 6.17: TXL rule that resolves subscripts

```
rule resolveSubscript
  replace [factor]
    SubName [factor]
  deconstruct SubName
    Llon [llon] '[ Exp [expression] ']
  construct MemCell [opt memoryCell]
    _ [getMemoryEntry Llon]
  deconstruct * MemCell
    Llon MemInfo [memoryInfo]
  deconstruct * [opt indexBounds] MemInfo
    LowBound [integernumber] UppBound [integernumber]
  construct NewExp [expression]
    Exp [evaluateExpression]
      [arrayOutOfBoundsError LowBound UppBound SubName]
  deconstruct NewExp
    Index [integernumber]
  deconstruct * [lexicalLevel] Llon
    LexLevel [integernumber]
  deconstruct * [orderNumber] Llon
    OrdNum [integernumber]
  deconstruct * [opt id] Llon
    Id [opt id]
  construct NewLlon [llon]
    '%( LexLevel , OrdNum [+ Index] [- LowBound] ) Id
  by
    NewLlon
end rule
```

Listing 6.18: TXL rule that replaces variable references with their values
6.3.2 Statements

Assignment statements are executed by replacing the value in the memory entry of the variable with the value obtained after evaluating the expression in the statement. The TXL function that executes an assignment statement is shown in Listing 6.19.

**Listing 6.19**: TXL function that executes an assignment statement

```txl
function executeAssignmentStmt
    match [statement] 
    construct Name [optSubscriptedName]
        Llon OptSub
    construct NewName [optSubscriptedName]
        Name [replaceSubScriptedVarsWithLlon]
    % resolve array subscripts, replace variables with their % values and evaluate the expression
    construct NewExp [expression]
        Exp [replaceSubScriptedVarsWithLlon]
        [getVariableValuesFromMemory]
        [evaluateExpression]
    deconstruct NewExp
        NewVal [factor]
    deconstruct NewName
        NewLlon [llon]
    import Memory [repeat memoryFrame]
    export Memory
        Memory [replaceValueInMemory NewLlon NewVal]
end function
```

Before the statements are executed all the procedure definitions in the program source are extracted and stored as a sequences of items of type [routine] in the global variable Procedures.

**Listing 6.20**: TXL rule that removes procedures and adds them to a sequence

```txl
rule createListOfProcedures
    replace $ [routine]
        Proc [routine]
    import Procedures [repeat routine]
    export Procedures
        Proc Procedures
    by
end rule
```

When a call statement is executed a copy of the procedure being called is obtained from the global variable Procedures. A new frame is pushed onto the memory to represent the local scope.
of the procedure being called. Memory entries are added to the frame for each of the parameters in the procedure header and their values are replaced with the values of the arguments in the call. The addresses of the variable arguments in the call are added to the memory entries of the variable parameters as items of the type [varArgAddress]. Memory entries are then added for the variables declared within the procedure and the statements in the procedure body are executed. The frame added onto the memory is then popped off. The values in the memory entries of the variable arguments are replaced with the values of the variable parameters in this frame before the frame is discarded. This is an implementation of parameter passing by value reference whereas variable parameters in PT Pascal are passed by reference. Listing 6.21 shows the function that executes call statements.

```
function executeCallStmt
  match [statement]
    Id [id] Arg [opt arguments]
  construct Proc [opt routine]
    _ [getProcedureFromList Id]
  deconstruct * [opt parameters] Proc
    Params [opt parameters]
  construct NewArg [opt arguments]
    Arg [evaluateArguments Params]
  import Memory [repeat memoryFrame]
  export Memory
    Memory [pushFrame]
  construct NewProc [opt routine]
    Proc [putInArgValues NewArg]
  deconstruct NewProc
    'procedure Id _ [opt parameters] ;
      DecList [repeat declarations]
    'begin
      Stmts [repeat statementOptSemicolon]
    'end;
  construct NewDecList [repeat declarations]
    DecList [allocateMemory]
  construct NewStmts [repeat statementOptSemicolon]
    Stmts [executeStatement]
  construct ProcFrame [opt memoryFrame]
    _ [returnProcFrame]
  deconstruct ProcFrame
    'frame FrNum [integerNumber]
    MemCells [repeat memoryCell]
  import Memory
  export Memory
    Memory [replaceValueWithVarParmValue each MemCells]
end function
```

Listing 6.21: TXL function that executes a call statement

86
After the evaluation of the conditional expression in an if statement, the statements in the if block are executed if the expression is ‘true’. If it is ‘false’ the statements in the else block are executed. In the absence of an else block, the first statement after the if statement is executed. While and repeat statements are executed using recursive TXL functions. The recursion ends when the control expression becomes ‘false’ in the case of a while statement and when it becomes ‘true’ in the case of a repeat statement. Listing 6.22 shows the recursive TXL function that executes a repeat statement.

Listing 6.22: Recursive TXL function that executes a repeat statement

```
function executeRepeatStmt
    replace [statement]
        Stmt [statement]
    deconstruct Stmt
        'repeat
            StmtBlock [repeat statementOptSemicolon]
        'until CondExp [expression]
    construct NewStmtBlock [repeat statementOptSemicolon]
        StmtBlock [executeStatement]
    construct NewCondExp [expression]
        CondExp [replaceSubScriptedVarsWithLlon]
            [getVariableValuesFromMemory]
            [evaluateExpression]
    deconstruct NewCondExp
        'false
    by
        Stmt [executeRepeatStmt]
end function
```

A case statement is executed by evaluating the case expression to obtain the value of the case selector. The list of case labels in each case alternative is then searched until a match is found for the case selector. The statements within that case alternative are then executed. A runtime error message is emitted if the case selector does not match any of the labels in the case statement.
A call is made to the rewrite procedure in order to open a file for writing and results in the contents of the file being overwritten if the file exists. If the file does not exist it is created and then opened for writing. The TXL built-in function `fput` appends files that are already open but opens and overwrites files that are closed. If the file does not exist `fput` creates it before writing to it. A call to the rewrite procedure is therefore executed by closing the file to be overwritten using the TXL built-in function `fclose`. An empty string is written to the file using the `fput` function before `fclose` is called to avoid the error message that will result if a non-existent file is closed.
Listing 6.23: TXL function that executes a call to the PT Pascal procedure rewrite

A call to the write procedure is made when a text file or char file is being written to. It is executed by evaluating the write expression and then adding this value to the write buffer of the file or the standard output stream as appropriate. If the width of the output is specified and the length of the value of the write expression is less than that specified trailing white spaces are added to the value until it is of the required length. Listing 6.24 shows functions that extend the value of the write expression to width specified and then add it to the write buffer.
function addExpnWithWidthToWriteBuffer FileId [name]
    match [expnOptColonExpn]
        Exp [expression] ': Width [expression]
    construct NewExp [expression]
        Exp [replaceSubscriptedVarsWithLlon]
            [getVariableValuesFromMemory]
                [evaluateExpression]
                    [removeQuotes]
    construct NewWidth [expression]
        Width [replaceSubscriptedVarsWithLlon]
            [getVariableValuesFromMemory]
                [evaluateExpression]
    deconstruct NewWidth
        WNum [integernumber]
    construct CLit [charlit]
        _ [quote NewExp]
        export Spaces [charlit]
        _
        construct Length [integernumber]
            _ [# CLit]
                [makeSpaces WNum]
        import Spaces
    construct SpaceId [id]
        _ [unquote Spaces]
    construct NewCLit [charlit]
        CLit [quote SpaceId]
    construct WriteExp [expression]
        NewCLit
    construct NewWriteExp [expression]
        WriteExp [removeQuotes]
    deconstruct NewWriteExp
        Fac [factor]
    construct NewFac [factor]
        Fac [addToWriteBuffer FileId]
    end function

rule makeSpaces Width [integernumber]
    replace [integernumber]
        Length [integernumber]
    where
        Length [< Width]
    construct String [stringlit]
        " "
    construct SingleSpace [id]
        _ [unquote String]
    import Spaces [charlit]
    export Spaces
        Spaces [quote SingleSpace]
    by
        Length [+ 1]
end rule

Listing 6.24: TXL functions that evaluate write expressions and add them to write buffers
A call is made to the write_int procedure if the file to be written to is an integer file. When executing a call to write_int the value of the write expression is written directly to the file without first storing it in a buffer. The value is written using the TXL built-in function fput. The [eofValue] attribute in the file’s buffer is then changed to ‘0’ since the file is no longer empty.

Listing 6.25: TXL function that writes to an integer file

```
function writeToIntFile Exp [expression] FileId [name]
    match [opt actualFileName]
        FileName [stringlit]
    deconstruct Exp
        Num [integernumber]
    construct NewNum [integernumber]
        Num [fput FileName]
    import FileBuffers [repeat fileBuffer]
    export FileBuffers
        FileBuffers [setEofValueToZero FileId]
end function
```

The writeln procedure only operates upon text or char files. The writeln procedure writes a new line character to the file specified as an argument. If no argument is specified it writes a newline character to the standard output stream. The TXL built-in functions put and fput that are used to write to the console or a file respectively write the text of the variable to which they are applied followed by a new line character. When a call to the writeln procedure is encountered it is executed by writing the contents of the appropriate write buffer out to the console or file using put or fput.
The reset procedure opens a file for reading. A call to the reset procedure is executed by using the TXL function fclose to close the file since this will ensure that the file pointer is at the beginning of the file when it is opened by the TXL fget function for reading. If the file does not exist TXL generates an error message to that effect. Listing 6.27 shows the function that executes a call to the reset procedure. Since write buffers for text and char files are only flushed when a call to writeln is executed it is possible that the write buffers still contain data when a call to reset is made. Therefore the contents of the write buffer of the file are flushed to the file before executing the reset call so that the most recent version of the file is being read.
When a character is to be read from a text or char file a call is made to the read_char procedure. This call is executed by filling the read buffer by reading a line from the standard input stream or file if the read buffer is empty. The first character from the read buffer is then retrieved and stored in memory as the value of the variable argument in the call. Before executing a read call on the standard input stream however the contents of the write buffer for the standard output stream must be flushed to the console if it is not already empty. No call is made to reset before a read of the standard input stream is done which is why the standard output buffer can only be flushed at this point.

Listing 6.27: TXL function that executes a call to the reset procedure

Listing 6.28: TXL function that executes a call to read a character
A call is made to the procedure read_int when an integer is to be read from the standard input stream or a file. This call is executed by reading a line into the read buffer if the buffer is empty and removing an integer from the beginning of the read buffer. This is done by removing leading white spaces and then parsing all the non white space characters until the next white space into an integer number. Again the write buffer for the standard output stream is flushed before the call is executed if the integer is to be read from the standard input stream.

A call to the readln procedure is made if the file pointer is to be moved to the beginning of the next line in the file. A readln call is executed by emptying the read buffer of the file argument. When a read call is executed a line is read into the read buffer if it is empty. By emptying the read buffer when a readln call is executed the interpreter is forced to read the next line from the file when the next read call is executed which is the desired effect of moving the file pointer to the beginning of the next line.

Listing 6.29: TXL functions that execute a call to the readln procedure

This implementation of the execution phase serves as an example of how an imperative language can be interpreted using a transformation system. It illustrates how a scoped memory can easily be simulated in TXL. However a disadvantage of implementing the memory as a global table, with new frames added when new scopes are entered, is that the TXL program runs out of memory space when executing a program with many large arrays. The program will execute successfully
but the output is interspersed with warnings about the lack of memory space that are generated by the TXL engine.

TXL does not easily lend itself to the implementation of I/O operations since the built-in I/O functions in TXL are very basic. Some of the inherent difficulties can be avoided through the use of file buffers but complicated file I/O operations might still be too difficult to implement. A possible solution for this is the use of external functions to implement I/O operations. External functions are functions that are written in a different language that can be called within a TXL program.

TXL’s ability to arbitrarily traverse trees makes it easy to simulate the control flow in the program while executing it. The TXL interpreter is slower when compared to the existing PT Pascal compiler. The declarative style of TXL however makes the interpretations rules easy to understand which is important if it is to be used as an aid in teaching.

In this chapter the execution phase is discussed. Memory entries are created for the declared variables and file buffers are created for the files listed as program parameters. The statements in the source program are then interpreted and error messages are emitted if run-time errors are detected. The next chapter discusses the code generation phase and the implementation of the tcode simulator. The code generation phase is run after the semantic analysis phase if the intermediate source produced by the Semantic Analyser is not to be directly executed.
Chapter 7
Code Generation and Simulation

The execution phase discussed in the previous chapter is run if the intermediate code generated by
the semantic analysis phase is to be directly executed. If the source program is to be transformed
into tcode instructions and then executed the code generation phase is run instead. In the code
generation phase the intermediate source produced by the semantic analysis phase is transformed
into an equivalent tcode instruction sequence. No error messages are emitted during this phase
since the semantic analysis phase has already checked for semantic errors and run time error
checking can only be performed later during the code execution phase. Each tcode instruction is
emitted with a numerical label that is used by the code simulator to navigate the instruction
sequence when executing the instructions.

This chapter contains examples of the creation and use of union grammars and describes the tcode
instruction sequences generated for the declarations and statements in a source program. It shows
examples or rules that demonstrate how an expression tree can be converted into a postfix token
stream. In later sections it describes the structure of the abstract machine that tcode is designed to
be executed on and illustrates how a simple stack machine can be implemented using TXL
[Cordy05].

7.1 Grammar Overrides

A tcode instruction that is emitted by the Code Generator consists of an integer label followed by
a tcode and optional operands. The integer label is used to indicate the location of the instruction
in the code area. The operand is generally a literal value, a variable reference or a file identifier.

In TXL both the input into the TXL program and the transformed output from it must conform to
the TXL input grammar. In the code generation phase however the form of the output from the
transformations does not conform to the input grammar. This is overcome by creating a union
grammar that combines the grammar definitions for PT Pascal [Ross80] declarations and
statements with grammar definitions for tcode instruction sequences. TXL redefine statements can
be used to combine PT Pascal forms with tcode forms. An example of this is shown in Listing 7.1

96
where the definition for [declarations] is overridden to accommodate the tcode instructions that generated for declared variables and procedures.

<table>
<thead>
<tr>
<th>redefine declarations</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>end redefine</td>
</tr>
</tbody>
</table>

Listing 7.1: A TXL override of the definition for [declarations] in the Code Generator

### 7.2 Variable Allocation

The Code Generator emits tcode instructions for variable allocation based on the declarations present in the source program. All identifiers and lexical level order number addresses required in the code execution phase are emitted as operands in tLiteralAddress instructions. A tFileDescriptor tcode is emitted for each file variable listed as a program parameter. The address of a declared variable that is emitted is followed by a tAllocateVar tcode unless the variable is of type subrange or array.
Listing 7.2: TXL function that emits tcode sequence for a variable declaration

A tSubrangeDescriptor tcode is emitted for subrange variables. This is preceded by the lower and upper bounds that are emitted as operands of tLiteralInteger instructions and the lexical level order number address of the variable. For array variables the bounds of the array component (if it is a subrange) and the bounds of the array are emitted followed by a tArrayDescriptor tcode.

Listing 7.3: TXL function that emits tcode instructions for a subrange declaration

A tSubrangeDescriptor tcode is emitted for subrange variables. This is preceded by the lower and upper bounds that are emitted as operands of tLiteralInteger instructions and the lexical level order number address of the variable. For array variables the bounds of the array component (if it is a subrange) and the bounds of the array are emitted followed by a tArrayDescriptor tcode.
7.3 Procedure Code Generation

When a procedure definition is encountered in the program source a tSkipProc instruction is generated. The tSkipProc instruction takes the label of the first instruction after the end of the procedure as an operand. Since the tSkipProc instruction is generated before the code sequence for the rest of the procedure is generated a placeholder label is generated as the operand for the tSkipProc instruction. This placeholder label is replaced with the actual label after all the code for the procedure has been generated.

The tSkipProc instruction is followed by a tEnter instruction with the lexical level of the procedure as an operand. This is used to indicate that a new scope has been entered. The procedure identifier and the label of the tEnter instruction are added to a table that contains the
locations of the entries into the procedures for which code has already been generated. This table is later used when code is generated for calls to the procedure.

Code is then generated for each of the parameters listed in the procedure header. For a variable parameter the lexical level order number address of the parameter is emitted as the operand of a tLiteralAddress instruction followed by a tStoreParmAddress instruction. For a value parameter the tLiteralAddress instruction that is generated is followed by a tStoreParmInteger, tStoreParmChar or tStoreParmBoolean instruction depending on the type of the parameter.

After code has been generated for the parameters of the procedure tcode instruction sequences are generated for the declarations and statements within the body of the procedure. This is followed by a tReturn instruction with the lexical level of the procedure as an operand, which indicates that the scope is exited, and a tProcedureEnd instruction. Listing 4 shows the TXL function that generates code for a procedure. Figure 7.2 shows the tcode instruction sequence that is generated for a procedure.
program myprog (output);
procedure myproc (var % (1, 0) a : char; % (1, 1) b : char);
begin
  % (1, 0) a VarParameter char := % (1, 1) b Variable char
end;
begin
end.

Figure 7.2: Example of code that is generated for a procedure
Listing 7.4: TXL function that generates code for a procedure

7.4 Statement Code Generation

Once code has been generated for all the declared variables and procedures in the source program code is generated for the statements.

7.4.1 Expression Code Generation

In the code generation phase the expressions are converted into postfix form when code is generated for them. Integers, characters and boolean values in expressions are emitted as operands for the tLiteralInteger, tLiteralChar and tLiteralBoolean instructions respectively. PT Pascal does not have a string data type but does allow strings as arguments in calls to the write procedure. When a string is encountered in an expression a tLiteralString instruction is emitted with the length of the string and the string itself as operands. This is then followed a tStringDescriptor instruction with the location of the string as an operand. Listing 7.5 and 7.6 show the functions that emit tcode instructions for a literal character and a string in an expression respectively.
A `tLiteralAddress` instruction accompanied by the lexical level order number address of the variable being referenced is emitted for each variable reference. This is then followed by a subscript code sequence in the case of subscripted variable references. Since variable references need to be replaced by the values of the variables before an expression can be evaluated a fetch code is then generated.
If the variable reference is subscripted a tSubscriptBegin instruction is emitted followed by the instruction sequence for the subscript expression. A tSubscriptAddress instruction is then emitted if the variable being referenced is a variable parameter. If it is not a tSubscriptInteger, tSubscriptChar or tSubscriptBoolean instruction is generated depending on the component type of the array.

For variables of type integer, char or Boolean the instructions tFetchInteger, tFetchChar or tFetchBoolean are emitted. For variable parameters a tFetchAddress is generally emitted. However in the case of variable parameters of the type array any reference to them will be subscripted and a tSubscriptAddress instruction will already have been generated. Since the fact that the symbol being referenced is a variable parameter as opposed to a variable is already dealt with when the tSubscriptAddress instruction is executed it is not necessary to follow it with a tFetchAddress instruction. Instead a tFetchInteger, tFetchChar or tFetchBoolean instruction is generated depending upon the component type of the array variable being referenced. Listing 7.7 shows the functions that generate the appropriate fetch codes for variable parameter references.
Listing 7.7: TXL functions that generate fetch codes for variable references

Figure 7.3 shows the tcode sequence that is generated for literal values and variable references in expressions.
In PT Pascal a term in an expression consists of a factor followed by an optional sequence of operator factor pairs. In order to convert a term to postfix form when code is being generated for
it code is generated for the first factor. For each of the operator factor pairs that follow, code is first generated for the factor and then for the operator. This results in the term being converted to postfix form. Listing 7.8 shows the functions that convert a term into postfix form and generate code for it.

Listing 7.8: TXL functions that convert a term into postfix form and generate code for it

```txl
function emitTerm
    replace [term]
        Fac [factor] bOpFacs [repeat binaryOperatorFactor]
    construct NewFac [factor]
        Fac [emitFactor] % make sure to run the rules in the
        % correct order so the code labels
        % are in sequence
    construct Postfix [repeat factorBinaryOperator]
    by
        NewFac Postfix
end function

function makeFactorPostfix OpFac [binaryOperatorFactor]
    replace [repeat factorBinaryOperator]
        FBOps [repeat factorBinaryOperator]
    deconstruct OpFac
        FOp [factorOperator] Fac [factor]
    construct Op [operator]
        FOp
    construct NewFBOp [factorBinaryOperator]
        Fac [emitFactor] Op [emitMultiply]
            [emitDivide]
            [emitModulus]
            [emitAnd]
    by
        FBOps [.. NewFBOp]
end function

function emitFactor
    replace [factor]
        Fac [factor]
    by
        Fac [removeBracketsAndEmitCode]
            [emitFac]
end function
```

Code is also generated for simple expressions and by first generating code for the first term in the simple expression and then generating code for the term in each operator term pair before generating code for the operator. Code is generated for the simple expressions in an expression
before it is generated for the operator thereby converting the entire expression into postfix form.
Listing 7.9 shows the functions that generates code for an expression. Figure 4 shows an example of the code generated for an expression.

```
function emitPostfixSimpExpression
   replace [expression]
       SExp [simpleExpression]
   by
       SExp [emitSimpExp]
end function

function emitPostfixSimpExpressionWithOperator
   replace [expression]
   by
       construct Op [operator] SOp
   by
       SExp1 [emitSimpExp] SExp2 [emitSimpExp]
       Op [emitEQ]
       [emitNE]
       [emitLT]
       [emitLE]
       [emitGT]
       [emitGE]
end function
```

Listing 7.9: TXL functions that generate code for an expression
7.4.2 Statement Code Generation

For assignment statements a tAssignBegin instruction is first emitted. This is followed by the lexical level order number address of the variable being assigned to as a tLiteralAddress instruction. If the variable being assigned to is an array variable it will be subscripted. Code is generated for this subscript in the same manner as it is for subscripted variable references in expressions. Code for the variable is followed by code that is generated for the expression in the assignment statement. Once this has been done an assign code is generated. If the variable being assigned to is of type integer, char or Boolean a tAssignInteger, tAssignChar or tAssignBoolean code is emitted. In the case of a variable parameter a tAssignAddress code is generated. However, if the variable parameter is of type array a tAssignInteger, tAssignChar or tAssignBoolean instruction is generated depending on the component type of the array.

```
program myprog (output);
  var
    % (0, 0) i : Boolean;
  begin
    % (0, 0) i Variable Boolean := 1 * 6 + 4 <= 10
  end.
```

Figure 7.4: An example of the code generated for an expression
The tcod instruction sequence that is generated for a call statement begins with the tCallBegin instruction. This is then followed by the code generated for each of the arguments. Since tcod is designed to be executed on a stack machine it is necessary to emit code for the arguments in the reverse order in which they appear in the source in order to have them appear in the correct order on the stack when they are being executed. For each value argument code is generated for the expression followed by a tParmEnd instruction. For a variable argument code is generated for the variable reference followed by a tVarParm instruction and a tParmEnd instruction. Once code has been generated for all the arguments listed in the call statement a tCallEnd instruction is generated with the location of the beginning of the procedure as an operand. The location of the procedure is obtained from the table that contains the locations of the procedures coupled with the procedure identifiers. Figure 7.5 shows an example of the code generated for a procedure call.

Listing 7.10: TXL function that generates code for an assignment statement

```txl
function emitAssignmentStmtCode
    replace [statement]
    import LabelNum [integer number]
    construct Label [integer number]
        LabelNum
    construct NextLabel [integer number]
        LabelNum [+ 1]
    export LabelNum
        LabelNum [+ 2]
    construct SubscriptCode [opt subscript Code]
        _ [getSubscriptCode OptSub Llon]
    construct NewExp [expression]
        Exp [emitPostfixExpression]
    construct AssignCode [opt assign Code]
        _ [getAssignCode Llon OptSub]
    by
        Label ': 'tAssignBegin
        NextLabel ': 'tLiteralAddress Llon
        SubscriptCode
        NewExp
        AssignCode
end function
```

Listing 7.10: TXL function that generates code for an assignment statement
program myprog (output);
var
  b : Boolean;
procedure greater (i : integer; var g : Boolean);
  begin
    g := i > 0
  end;
begin
  greater (3, b)
end.

Figure 7.5: Example of code generated for a call statement
For if statements a tIfBegin instruction is first generated followed by the code that is generated for
the conditional expression. This is then followed by a tIfThen instruction that takes the location
of the start of the else block as an operand. In the absence of an else clause this will be the same
as the location of the tIfEnd instruction that is generated when the end of the if statement is
reached. Code is then generated for the statements within the if block. If an else clause is not
present the tIfEnd instruction is emitted. If there is an else clause a tIfMerge instruction is emitted
with a temporary label as its operand. This placeholder operand is later replaced with the location
of the tIfEnd instruction. Code is then generated for the statements within the else block. This is
then followed by a tIfEnd instruction.

```plaintext
program myprog (output);
var
  % (0, 0) b : Boolean;
begin
  if 0 < 1 then
    % (0, 0) b Variable Boolean := true
  else
    % (0, 0) b Variable Boolean := false
end.
```

Figure 7.6: Example of the code generated for an if statement
A tWhileBegin instruction is generated at the beginning of a while statement. This is followed by the code for the control expression. After the code for the control expression is generated a tWhileTest instruction is generated with the location of the first statement after the while statement as an operand. This is followed by code for the statements in the while block. The last instruction that is generated for the while statement is the tWhileEnd statement which has the location of the tWhileBegin instruction as its operand.

Listing 7.11 shows the TXL function that generates code for a while statement.

```
function emitWhileStmtCode
  replace [statement]
    'while Exp [expression] 'do
    StmtList [statementOptSemicolon]
  construct Stmts [repeat statementOptSemicolon]
    StmtList
  import LabelNum [integernumber]
  construct WhileBeginLoc [integernumber]
    LabelNum
  export LabelNum
    LabelNum [+ 1]
  construct NewExp [expression]
    Exp [emitPostfixExpression]
  import LabelNum
  construct NextLabel [integernumber]
    LabelNum
  export LabelNum
    LabelNum [+ 1]
  construct NewStmts [repeat statementOptSemicolon]
    Stmts [emitStmtCode]
  import LabelNum
  construct WhileEndLoc [integernumber]
    LabelNum
  export LabelNum
    LabelNum [+ 1]
  by
    WhileBeginLoc ': 'tWhileBegin
    NewExp
    NextLabel ': 'tWhileTest WhileEndLoc [+ 1]
    NewStmts
    WhileEndLoc ': 'tWhileEnd WhileBeginLoc
end function
```

Listing 7.11: TXL function that generates code for a while statement
For a repeat until statement a tRepeatBegin instruction is first emitted followed by the code for the statements within the repeat until block. This is followed by the tRepeatControl instruction. The code for the control expression in the repeat until statement is then generated. This is followed by a tRepeatTest instruction that takes the location of the tRepeatBegin instruction as an operand. Figure 7.7 shows an example of the code generated for a repeat until statement.

```
program myprog (output);
var
  % (0, 0) i : integer;
begin
  % (0, 0) i Variable integer := 1;
  repeat
    % (0, 0) i Variable integer := % (0, 0) i Variable integer + 1
  until % (0, 0) i Variable integer = 10
end.
```

Figure 7.7: Example of the code generated for a while statement

For case statements a tCaseBegin instruction is emitted followed by the code for the case selector expression. A tCaseSelect instruction with the location of the case table as an operand is the
emitted. The case table is a table in which the location of each alternative in the case statement is stored along with the case labels for that alternative. The case table is used to decide which instruction needs to be executed next based on the value of the case selector. Code is next generated for each of the case alternatives in the order in which they appear in the source program. For each case alternative code is generated for the statements in the alternative followed by a tCaseMerge instruction label which takes the location of the tCaseEnd instruction as its operand. Since the location of the tCaseEnd instruction is not known at this point a placeholder label is inserted and is later replaced by the actual location of the tCaseEnd instruction. As code is generated for each alternative the location of the beginning of the alternative along with the case labels for that alternative are entered into the case table. Once code has been generated for all the case alternatives a tCaseTable instruction is emitted with the number of case alternatives and the case table itself as operands. Finally a tCaseEnd instruction is emitted. An example of the code generated for a case statement is shown in Figure 7.8.
A `tWriteBegin` instruction is first generated for a call to the write procedure. This is followed by code for the write expression and the width expression. If no width is specified a default width of 1 is emitted as an operand for a `tLiteralInteger` instruction. This is then followed by a `tLiteralAddress` instruction with the identifier of the file being written to (output in the case of the standard output stream) as an operand. A `tTrap` instruction is then generated with the operand `trWriteChar`. This indicates that the value from the write expression is to be written to a character.
or text file. Two tPopStack instructions and a tParmEnd instruction are then generated. I/O operations are generally implemented by calling traps. Once the trap is called the arguments to the trap need to be popped off the stack which is why the tPopStack instructions are generated.

The same code that is generated for a call to the write procedure is generated for a call to write_int with a few changes. The width cannot be specified for a write to an integer file so default width of 1 is emitted for the sake of symmetry. The operand to the tTrap instruction is trWriteInt in this case indicating that an integer file is being written to. Figure 7.9 shows the code generated for calls to write and write_int.

```
program myprog (output, intfile);
  var
    intfile : file of integer;
  begin
    write (output, 'Write to output');
    rewrite (intfile);
    write_int (intfile, 23);
  end.
```

```
0 : tLiteralAddress output
1 : tFileDescriptor
2 : tLiteralAddress intfile
3 : tFileDescriptor
4 : tWriteBegin
5 : tLiteralString 15 'Write to output'
6 : tStringDescriptor 5
7 : tLiteralInteger 1
8 : tLiteralAddress output
9 : tTrap trWriteChar
10 : tPopStack
11 : tPopStack
12 : tParmEnd
13 : tLiteralAddress intfile
14 : tTrap trRewrite
15 : tWriteBegin
16 : tLiteralInteger 23
17 : tLiteralInteger 1
18 : tLiteralAddress intfile
19 : tTrap trWriteInt
20 : tPopStack
21 : tPopStack
22 : tParmEnd
23 : tTrap trHalt
```

Figure 7.9: Example of code generated for calls to write and write_int
For calls to the read_char and read_int a tReadBegin instruction is generated. This is followed by
the code that is generated for the read variable. The identifier of the file that is being read from
(input in the case of the standard input stream) is then emitted as an operand to a tLiteralAddress
instruction. A tTrap instruction is then generated with a trReadChar operand for calls to read_char
and a trReadInt operand for calls to read_int. This is then followed by a tPopStack instruction and
a tParmEnd instruction. A TXL function that generates code for a call to read_char is shown in
Listing 7.12.

Listing 7.12: TXL function that generates code for a call to read_char

Calls to the reset, rewrite, readln and writeln procedures result in the same tcode instruction
sequence. The file identifier is first emitted as an operand to a tLiteralAddress instruct followed
by a tTrap instruction with the appropriate operand; trReset, trRewrite, trReadln and trWriteln in
the case of a call to reset, rewrite, readln and writeln respectively. Figure 7.10 shows code that is
generated for a call to writeln.
A tTrap instruction with trHalt as an operand is emitted as the last instruction after instructions have been generated for the entire program.

7.5 Code Simulation

Tcode instructions are designed for execution on a stack machine. The Code Simulator is a TXL implementation of the abstract machine that executes the tcode instruction sequence produced by the Code Generator. The Code Simulator makes use of the same memory and file buffer structures that are used in the execution phase.

7.5.1 Abstract Machine Structure

All the data structures used to implement the abstract machine such as the expression stack, the instruction pointer, the return stack, the memory and the file buffers are declared as global variables since they must be accessible to all the rules in the Code Simulator. The main function in which these global variables are initialised is shown in Listing 7.13.
Listing 7.13: TXL main function in the Code Simulator that initialises the global variables

The abstract machine on which tcode is executed consists of an expression stack in which operands and results from operations performed are stored. The expression stack is implemented as a global variable Stack that is a sequence of items of the type [operand]. Listing 7.14 shows the grammar definition for [operand].
The abstract machine has an instruction pointer that contains the location of the next instruction that is to be executed. It is implemented as the global variable IP of the type [integernumber] since integer labels are used to specify the location of instructions. It also has a return stack which is used to store the location of the instruction that is to be executed after a call to a procedure results in the procedure’s code being executed. The return stack is implemented as the global variable ReturnStack that contains a sequence of items of the type [integernumber]. The registers used by the machine are implemented as local variables that are constructed within rules as needed. The register variables are optional items of type [operand] and are named StackRight, StackLeft and StackDest in all the rules for the sake of consistency. Listing 7.15 shows a TXL function in which operands that are popped off the stack are stored in registers.

Listing 7.14: TXL grammar definition of [operand]

```text
define operand
    | [integernumber]
    | [charlit]
    | [booleanValue]
    | [name]
end define
```
The same memory structure and file buffer structure are used by the Code Simulator as the ones used in the execution phase with the only notable difference being that the Code Simulator retrieves the information needed when creating memory entries such as the lexical level order number address of the variable from the stack whereas this same information was available as part of the declaration that was matched by a rule in the execution phase. When creating a file buffer the file identifier is also obtained by popping it off the stack where is has been placed when a previous instruction was executed. Listing 7.16 shows a TXL function that creates a memory entry for a variable in the code simulation phase. A TXL function that creates a file buffer is shown in Listing 7.17.
Listing 7.16: TXL function in the Code Simulator that creates a memory cell for a variable

```plaintext
function executeTAllocateVarInst
  match [instruction]
    'tAllocateVar
    construct StackRight [opt operand]
      _ [popStack]
    deconstruct StackRight
    Llon [llon]
    construct NewCell [memoryCell]
      Llon 'no_value
    import Memory [repeat memoryFrame]
    export Memory
    Memory [addMemoryCell NewCell]
end function
```

Listing 7.17: TXL function in the Code Simulator that creates a file buffer

```plaintext
function executeTFileDescriptorInst
  match [instruction]
    'tFileDescriptor
    construct StackRight [opt operand]
      _ [popStack]
    deconstruct StackRight
    FileId [name]
    construct NewFileId [name]
      FileId [createInputBuffer]
      [createOutputBuffer]
      [createOtherBuffer]
end function
```

7.5.2 Instruction Execution

The tcode instruction sequence produced by the Code Generator is parsed into a sequence of items of the type [tCodeInstruction]. The TXL grammar definitions of [program] and [tCodeInstruction] are shown in listing 7.18. An item of type [tCodeInstruction] consists of an item of type [label] followed by an item of type [instruction]. A label is an integer followed by the character ‘:’. An instruction is either a tcode followed by optional operands or a case table. Listing 7.19 shows the TXL grammar definition of [label] and [instruction].
The location of the instruction to be executed is in the global variable IP. IP is initialised to ‘0’ since the first instruction in the tcode instruction sequence is always given the label ‘0 :’ by the Code Generator. The instruction to be executed is found by performing a searching deconstruct on the [program] tree which is a sequence of items of type [tCodeInstruction]. Once the instruction whose label matches the IP is found the IP is updated by incrementing it. Functions are then applied to instruction which each match a different tcode. When the tcode in the instruction matches the tcode in the pattern of one of the functions applied to the instruction the instruction is executed. The rule that performs the tasks of finding the right instruction, updating the IP and then applying the execution functions to the instructions only ends when the haltExecution rule succeeds.
Listing 7.20: TXL rule in the Code Simulator that executes the instructions

```plaintext
rule executeTCodeInstructions
    replace [program]
        P [program]
    import IP [integer number] % location of the instruction to be % executed
deconstruct * [tCodeInstruction] P
    IP ': Inst [instruction]
export IP
    IP [+ 1] % location of the next instruction to be % executed
construct NewInst [instruction]
Inst [executeTLiteralAddressInst] [executeTLiteralCharInst]
    [executeTLiteralBooleanInst] [executeTLiteral StringInst]
    [executeTFileDescriptorInst] [executeTAllocateVarInst]
[executeTSubrangeDescriptorInst]
    [executeTArrayComponentBoundsInst]
[executeTArrayDescriptorInst] [executeTSkipProcInst]
[executeTEnterInst] [executeTStoreParmAddressInst]
[executeTStoreParmIntegerInst]
[executeTStoreParmCharInst]
[executeTStoreParmBooleanInst] [executeTExitInst]
[executeTProcedureEndInst] [executeTFetchAddressInst]
[executeTFetchIntegerInst] [executeTFetchCharInst]
[executeTFetchBooleanInst] [executeTMultiplyInst]
[executeTArrayElementInst] [executeTModulusInst]
[executeTAddInst] [executeTSubtractInst] [executeTEqInst]
[executeTNeInst] [executeTGtInst] [executeTGeInst]
[executeTLtInst] [executeTLeInst] [executeTAndInst]
[executeTOrInst] [executeTNegateInst] [executeTNotInst]
[executeTChrInst] [executeTOrdInst] [executeTEofInst]
[executeTEolnInst] [executeTAssignAddressInst]
[executeTAssignIntegerInst] [executeTAssignCharInst]
[executeTAssignBooleanInst]
[executeTSubscriptAddressInst]
[executeTSubscriptIntegerInst]
[executeTSubscriptCharInst]
[executeTSubscriptBooleanInst]
[executeTCallEndInst] [executeTIfThenInst]
[executeTIfMergeInst] [executeTCaseSelectInst]
[executeTCaseTableInst] [executeTCaseMergeInst]
[executeTWhileTestInst] [executeTWhileEndInst]
[executeTRepeatTestInst] [executeWriteCharTrap]
[executeWriteIntTrap] [executeWriteIntTrap]
[executeRewriteTrap] [executeReadCharTrap]
[executeReadInt Trap] [executeReadlnTrap]
[executeResetTrap] [haltExecution]
    by
end rule
```

125
The haltExecution function matches a tTrap instruction with trHalt as the operand and stops execution by calling the built-in TXL function quit. Listing 7.20 shows the rule that executes the instruction in the Code Simulator and listing 7.21 shows the haltExecution function.

<table>
<thead>
<tr>
<th>function haltExecution</th>
</tr>
</thead>
<tbody>
<tr>
<td>match [instruction]</td>
</tr>
<tr>
<td>'tTrap 'trHalt</td>
</tr>
<tr>
<td>import FileBuffers [repeat fileBuffer]</td>
</tr>
<tr>
<td>export FileBuffers</td>
</tr>
<tr>
<td>construct Quit [id]</td>
</tr>
<tr>
<td>_ [quit 0]</td>
</tr>
<tr>
<td>end function</td>
</tr>
</tbody>
</table>

Listing 7.21: TXL function in the Code Simulator that halts execution

Implementing the code generation phase in TXL illustrates how a variety of tasks can be performed using a source transformation system. The description of the grammar overrides necessary in order to transform the typed intermediate source produced by the Semantic Analyser into a tcode instruction sequence provide an example of a union grammar. If used in teaching it will provide students with an opportunity to experiment with grammar programming. Grammar overrides are necessary due to the strongly typed nature of TXL. While TXL’s strong typing may introduce certain constraints when perform certain compilation tasks it has the advantage of ensuring that the output from the code generator is always well formed.

The declarative style of TXL patterns and replacements serves to demonstrate the obvious correspondence between the language constructs in PT Pascal and their equivalent tcode instruction sequences. This improves the ability to comprehend the tasks being performed by the Code Generator. This implementation also provides an example of how an expression tree can be converted into a postfix token stream using TXL.

The implementation of the Code Simulator using TXL demonstrates how a stack machine can be implemented using TXL. The Code Simulator however has the same problem as the execution phase and runs out of memory when executing a program with large arrays. This is expected since the execution phase and the code simulation phase share the same memory module. The code
simulator is slower than the execution phase because of the searching deconstruct that needs to be done to find each instruction before executing it and because the Code Simulator is organised such that an attempt is made to apply every possible rule to an instruction even after one of the rules has been successfully applied.

In the code generation phase described in this chapter the source program is converted into a tcode instruction sequence with expressions converted into postfix form. The tcode instruction sequence generated by the Code Generator is then executed in the code simulation phase. The Code Simulator is a simple stack machine implemented in TXL with the help of global variables. In the next chapter the TXL-based PT Pascal compiler/interpreter is evaluated and certain aspects of it are compared against the existing PT Pascal compiler.
Chapter 8
Evaluation

The previous chapters describe the implementation of the phases in the PT Pascal [Ross80] interpreter. In this chapter the TXL [Cordy05] implementation of the PT Pascal compiler/interpreter is evaluated both qualitatively and quantitatively. The qualitative evaluation takes into consideration aspects such as ease of comprehension and use while the quantitative evaluation focuses on measures such as execution speed and size of code. Both the advantages and the limitations of the approach are discussed and the TXL implementation is compared against the existing S/SL implementation of the compiler.

8.1 Qualitative Evaluation

As with any technology there are both advantages and disadvantages to implementing the PT Pascal compiler/interpreter using TXL. There are many advantages to implementing a language using TXL if the objective is to use it as a teaching tool. The declarative style of TXL makes its rules easy to comprehend and makes explicit the relationship between the input into a compiler phase and the output generated as a consequence. TXL was designed to as a rapid prototyping tool for experiments in language design. Extending a language that is implemented in TXL is easy to do.

For example, if PT Pascal were to be extended to allow for an optional sequence of elsif clauses in the if statement the TXL-based PT Pascal compiler/interpreter would only require two additions; a grammar definition of the new form of the if statement that permits elsif clauses and a rule that transforms elsif clauses into nested if else statements. The additions to the grammar include a keys statement that declares elsif a keyword, a redefine of the [ifStatement] nonterminal to allow elsif clauses and a grammar definition that specifies the form of an elsif clause. Listings 8.1 and 8.2 show the original grammar definition of [ifStatement] and the changes that need to be made to the PT Pascal grammar respectively. Listing 8.3 shows the original main function of the Parser. The rule that transforms elsif clauses into nested if else statements must be added to the parser as shown in Listing 8.4.
define ifStatement
    'if [expression] 'then [NL] [IN]
    [statement] [NL] [EX]
    [opt elseClause]
end define

define elseClause
    'else [NL] [IN]
    [statement] [NL] [EX]
end define

Listing 8.1: Original grammar definition for if statement

define elsifClause
    'elsif [expression] 'then [NL] [IN]
    [statement] [NL] [EX]
end define

Listing 8.2: Additions to the grammar to permit elsif clauses

include "PTPascal.Grm"

function main
    match [program]
        P [program]
end function

Listing 8.3: Original main function of the Parser
The changes that need to be made to the S/SL implementation of the PT Pascal compiler to allow elsif clauses also involves declaring elsif as a keyword and converting an elsif clause into a nested if else statement. The keyword elsif must first be entered into the file containing the standard identifiers that is used by the screener routine. A parser token for elsif must be added to the output token class of the S/SL program for the scanner and to the input token class of the S/SL program for the parser. The input and output tokens are assigned numerical values in the PT Pascal program that implements the scanner/screener and parser routines. These values must also be updated.

The rule in the S/SL parser that parses an if statement must also be modified to accept an elsif token and then emit the semantic token sequence for a nested if else statement. Listings 8.5 and 8.6 show the original S/SL rule that generates the semantic tokens for an if statement and a
modified rule that emits nested if else token sequences when an elsif token is matched respectively.

```
IfStmt : 
  .sIfStmt
  @Expression
  .sExpnEnd
  'then' .sThen
  @Statement
  |
  | 'else':
  | .sElse
  | @Stmt
  | '*':
};
```

Listing 8.5: Original S/SL rule that generates semantic tokens for an if statement

```
IfStmt : 
  .sIfStmt
  @Expression
  .sExpnEnd
  'then' .sThen
  @Statement
  {[
    | 'elsif':
      .sElse
      @IfStmt
    | 'else':
      .sElse
      @Stmt
      >
    | '*':
      >
  ]};
```

Listing 8.6: Modified S/SL rule that accepts elsif and emits nested if else semantic tokens

Another advantage of the TXL implementation is that any modifications made can be tested immediately by running the transform that was modified. In the S/SL implementation both the S/SL program and its PT Pascal implementation must be recompiled after modifications are made before it can be tested. In the PT Pascal compiler/interpreter all the implementation is done purely in TXL thereby avoiding the complexities of using different technologies and interfacing between
them. Making modifications to the TXL implementation will result in exposure to source transformation technology and an opportunity to use new programming paradigms.

The TXL implementation also has certain limitations. Since programming in TXL is a shift in paradigm the learning curve can be steep. Line numbers cannot be easily handled in TXL so the error messages can not be accompanied by the appropriate line number. Also TXL is not representative of the technologies used to build production compilers and it might be useful for students to become accustomed to those technologies. Other limitations are of a quantitative nature and are discussed in the next section.

8.2 Quantitative Evaluation

In terms of quantitative measures such the speed the TXL implementation does very poorly in comparison to the S/SL implementation. It is expected that the TXL implementation is slower since it is interpreted. Moreover each phase is implemented as a separate TXL program. The execution of each TXL program is done in three phases; the input into the program is parsed, the transformations are applied to the parse tree and the transformed tree is unparsed to provide the output text. This is more time consuming than a traditional implementation where parsing is only done once.

For the purpose of the quantitative evaluation a set of PT Pascal programs were executed using both the TXL-based PT Pascal compiler/interpreter and the traditional PT Pascal compiler. These programs are part of the set of programs used to test various PT Pascal language features in the introductory compiler course at Queen’s University. Table 8.1 on the following page contains brief descriptions of the programs used for theses tests.

The tests were conducted on a SPARC machine running SunOS 5.10 on four CPUs. Each program was compiled and executed five times and the times were then averaged to obtain the results. The times were measured in seconds using the UNIX time command and the system and user time returned by the command are listed in Table 8.2. The system time is the time spent executing the process in the kernel of the operating system and the user time is the time spent executing the process outside the kernel. The total time that a process takes to execute is a sum of the system and user times.
<table>
<thead>
<tr>
<th>Description</th>
<th>Number of lines of code</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarytree.pt</td>
<td>53</td>
</tr>
<tr>
<td>bubble.pt</td>
<td>62</td>
</tr>
<tr>
<td>bust.pt</td>
<td>414</td>
</tr>
<tr>
<td>cache.pt</td>
<td>190</td>
</tr>
<tr>
<td>pascal.pt</td>
<td>38</td>
</tr>
<tr>
<td>primes.pt</td>
<td>56</td>
</tr>
</tbody>
</table>

Table 8.1: Description of test programs

The time taken by the TXL-based PT Pascal compiler/interpreter to execute a set of PT Pascal programs is compared against the time taken by the existing PT Pascal compiler to compile the same programs and to execute the object files produced in Table 8.2.
Table 8.2: Comparison of TXL-based PT Pascal compiler/interpreter and traditional PT Pascal compiler execution times

<table>
<thead>
<tr>
<th></th>
<th>Compile Time</th>
<th>Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TXL-based PT Pascal Interpreter</td>
<td>Traditional PT Pascal Compiler</td>
</tr>
<tr>
<td></td>
<td>TXL-based PT Pascal Compiler</td>
<td>Traditional PT Pascal Compiler</td>
</tr>
<tr>
<td>binarytree.pt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>system time</td>
<td>0.032s</td>
<td>0.054s</td>
</tr>
<tr>
<td>user time</td>
<td>0.048s</td>
<td>0.030s</td>
</tr>
<tr>
<td>total time</td>
<td>0.080s</td>
<td>0.084s</td>
</tr>
<tr>
<td>bubble.pt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>system time</td>
<td>0.030s</td>
<td>0.056s</td>
</tr>
<tr>
<td>user time</td>
<td>0.042s</td>
<td>0.030s</td>
</tr>
<tr>
<td>total time</td>
<td>0.072s</td>
<td>0.086s</td>
</tr>
<tr>
<td>bust.pt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>system time</td>
<td>0.046s</td>
<td>0.026s</td>
</tr>
<tr>
<td>user time</td>
<td>0.210s</td>
<td>0.118s</td>
</tr>
<tr>
<td>total time</td>
<td>0.256s</td>
<td>0.144s</td>
</tr>
<tr>
<td>cache.pt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>system time</td>
<td>0.044s</td>
<td>0.042s</td>
</tr>
<tr>
<td>user time</td>
<td>0.112s</td>
<td>0.068s</td>
</tr>
<tr>
<td>total time</td>
<td>0.156s</td>
<td>0.110s</td>
</tr>
<tr>
<td>pascal.pt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>system time</td>
<td>0.030s</td>
<td>0.044s</td>
</tr>
<tr>
<td>user time</td>
<td>0.038s</td>
<td>0.034s</td>
</tr>
<tr>
<td>total time</td>
<td>0.068s</td>
<td>0.078s</td>
</tr>
<tr>
<td>primes.pt</td>
<td></td>
<td></td>
</tr>
<tr>
<td>system time</td>
<td>0.034s</td>
<td>0.060s</td>
</tr>
<tr>
<td>user time</td>
<td>0.044s</td>
<td>0.032s</td>
</tr>
<tr>
<td>total time</td>
<td>0.078s</td>
<td>0.092s</td>
</tr>
</tbody>
</table>

As the times listed in Table 8.2 show the time taken by the TXL-based PT Pascal compiler/interpreter to compile a program compares favourably in most instances against the time taken to compile a program by the traditional implementation of the PT Pascal compiler. The time taken to directly interpret the program however is much greater than the time taken to execute the
compiled code as expected. The execution times for the code generated by the traditional implementation of the PT Pascal compiler is zero seconds in many instances because the UNIX time command has a resolution of milliseconds and these programs execute in less than one millisecond.

Table 8.3 shows the size of the TXL-based PT Pascal compiler/interpreter and the size of the PT Pascal compiler in terms of the number of lines of code. The components are grouped by phase for ease of comparison.

<table>
<thead>
<tr>
<th>Component</th>
<th>TXL-based PT Pascal compiler/interpreter</th>
<th>Traditional PT Pascal compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lexical And Syntactic Analysis LOC</td>
<td>253</td>
<td>S/SL 723</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PT Pascal 1076</td>
</tr>
<tr>
<td>Semantic Analysis LOC</td>
<td>3466</td>
<td>S/SL 2285</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PT Pascal 1697</td>
</tr>
<tr>
<td>Total LOC for Lexical, Syntactic and Semantic analysis</td>
<td>3719</td>
<td>5781</td>
</tr>
<tr>
<td>Tcode Generation LOC</td>
<td>1973</td>
<td></td>
</tr>
<tr>
<td>SUN SPARC Assembly Code Generation LOC</td>
<td></td>
<td>S/SL 2517</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PT Pascal 1684</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Runtime routines 312</td>
</tr>
<tr>
<td>Execution LOC</td>
<td>1369</td>
<td></td>
</tr>
<tr>
<td>Code Simulation LOC</td>
<td>1665</td>
<td></td>
</tr>
<tr>
<td>Execution and Code Simulation shared modules LOC</td>
<td>621</td>
<td></td>
</tr>
</tbody>
</table>

Table 8.3: Number of lines of code in the TXL-based PT Pascal compiler/interpreter and traditional PT Pascal compiler

The semantic analysis phase in the S/SL implementation of the PT Pascal compiler performs the necessary semantic checks and converts the semantic token stream into a tcode instruction sequence. In the TXL-based PT Pascal compiler/interpreter however the semantic checking is
done in the semantic analysis phase and the output from the semantic analysis phase is transformed into a tcode instruction sequence by the code generation phase. The separation of these two phases increases the size of the TXL-based PT Pascal compiler/interpreter since the rules have to be written to traverse the program tree twice. Each phase requires a set of grammar overrides. The TXL-based PT Pascal compiler/interpreter also provides to options for execution, direct execution and the execution of tcode. Despite all these considerations the TXL-based PT Pascal compiler is still smaller than the existing PT Pascal compiler.

The direct interpreter and the code simulator in TXL-based PT Pascal compiler/interpreter do not scale well. As the size of the program and the size of the data structures within the program increase the amount of memory required by the execution phase or the code simulation phase also increases. Execution is completed even when the memory available is insufficient but speed is sacrificed in the process.

The limitations of the TXL-based PT Pascal compiler/interpreter show that it is not feasible to use it in real world situations without a great deal of tuning to optimise performance. However these limitations are of no great consequence in a teaching environment. Depending upon the objectives of the course the TXL implementation can be of much use. The next chapter concludes this thesis by summarising the work described so far and discussing future work.
Chapter 9
Conclusion

The objective of this thesis is to examine the feasibility of implementing a language using a source transformation system like TXL [Cordy05]. An interpreter has therefore been implemented for the PT Pascal [Ross80] programming language using TXL. The interpreter is implemented in phases based upon the traditional phase structure of compilers.

The lexical and syntactic analysis phase checks the source program for syntax errors. The semantic analysis phase creates a symbol table to store the attributes of the symbol and makes use of this information to perform semantic checks on the source program. The semantic analysis phase then transforms the source program by annotating variable references with type information. The typed source program can then either be directly executed in the execution phase or transformed into a tcode instruction sequence in the code generation phase. The tcode instruction sequence generated by the code generator is then executed in the code simulation phase.

The TXL-based PT Pascal compiler/interpreter provides a fully worked out example of how TXL can be used to perform various language implementation tasks. It can be used as a teaching tool to introduce compiler implementation concepts, source transformation technologies and a different programming paradigm at an introductory level. The declarative style of TXL makes the rules in the implementation easy to understand and makes the relationship between the input into and output from each phase more explicit. Since TXL is designed to aid in the rapid prototyping of language extensions modifications to the TXL-based PT Pascal compiler/interpreter can be made and tested easily. A further advantage to implementing the PT Pascal compiler/interpreter in TXL is that no other tools are required thereby eliminating the added complexity of interfacing between various tools such as S/SL, PT Pascal itself, an assembler and a C compiler that are necessary for the traditional implementation of the PT Pascal compiler.
As with all approaches implementing a language using TXL also has limitations. The most obvious limitation is the amount of time taken by the TXL-based PT Pascal compiler/interpreter to execute a program. Each of the phases in the TXL-based PT Pascal compiler/interpreter is implemented as a separate TXL program. A TXL program is executed in three phases namely the parsing phase, the transformation phase and the unparsing phase. As a consequence of this the source program is not only transformed as expected in each phase but also parsed and unparsed which results in an increase in the time taken to execute it.

Another limitation is that design of the execution phase and the code simulation phase is not scalable. As program size increases, the memory available to the transforms becomes insufficient and performance degrades. With respect to the use of the TXL-based PT Pascal compiler/interpreter as a teaching tool in introductory compiler courses, the hiding of the implementation details may be viewed as a disadvantage since almost all production compilers are implemented using traditional methods and an understanding of implementation details is advantageous when working with them. Therefore the advantages of the TXL-based implementation over the traditional implementation of the PT Pascal compiler are, to an extent, dependent on the focus of the course.

There are many directions for future work on this experiment. Further work can be done to tune the TXL programs in the TXL-based PT Pascal compiler/interpreter to make them more efficient and scalable by reducing the amount of memory they require. It must be noted however that tuning TXL rules for better efficiency and speed has a tendency to make them less comprehensible which would make the TXL-based PT Pascal compiler/interpreter less attractive as a teaching tool. At present the interpreter does not perform any optimisations on the source program. An optimisation phase can be added to the interpreter. A byte code generator that transforms the tcode instruction sequence into byte code and a byte code interpreter can be implemented in TXL. Since one of the motivations for this work is to use the TXL-based PT Pascal compiler/interpreter as a teaching tool in an introductory compiler construction course it will be of interest to perform a usability study to learn if students find it easier to understand compiler concepts when using the TXL-based implementation.
References


