The South African Institute of Computer Scientists
and
Information Technologists

Proceedings

of the

1996 National Research and Development Conference

Industry meets Academia

Interaction Conference Centre, University of Natal,
Durban.
26 & 27 September

Edited by
Vevek Ram

ISBN 0-620-20568-7

Cover printed by Natal Printers (Pty) Ltd, Pietermaritzburg
Copying by the Multicopy Centre, University of Natal, Pietermaritzburg
Binding by Library Technical Services, University of Natal, Pietermaritzburg

FOREWORD

This book is a collection of papers presented at the National Research and Development Conference of the Institute of Computer Scientists and Information Technologists, held on 26 & 27 September, at the Interaction Conference Centre, University of Natal, Durban. The Conference was organised by the Department of Computer Science and Information Systems of The University of Natal, Pietermaritzburg.

The papers contained herein range from serious technical research to work-in-progress reports of current research to industry and commercial practice and experience. It has been a difficult task maintaining an adequate and representative spread of interests and a high standard of scholarship at the same time. Nevertheless, the conference boasts a wide range of high quality papers. The program committee decided not only to accept papers that are publishable in their present form, but also papers which reflect this potential in order to encourage young researchers and to involve practitioners from commerce and industry.

The organisers would like to thank IBM South Africa for their generous sponsorship and all the members of the organising and program committees, and the referees for making the conference a success. The organisers are indebted to the Computer Society of South Africa (Natal Chapter) for promoting the conference among its members and also to the staff and management of the Interaction Conference Centre for their contribution to the success of the conference.

On behalf of the Organising Committee
Vevek Ram
Editor and Program Chair
Pietermaritzburg, September 1996

Organising Committee

Conference General Chairs
Mr Rob Dempster and Prof Peter Warren (UNP)

Organising Chair
Dr Don Petkov (UNP)

Secretariat
Mrs Jenny Wilson

Program Chair
Prof Vevek Ram (UNP)

Program Committee

Prof Peter Wentworth, Rhodes
Dr Milan Hajek, UDW
Prof Derek Smith, UCT
Prof Anthony Krzesinski, Stellenbosch
Dr Don Petkov, UNP
Mr Rob Dempster, UNP
Prof Peter Warren, UNP

Table of Contents

Foreword i

Organising Committee ii

List of Contributors vi

Keynote Speaker

The Role of Formalism in Engineering Interactive Systems 1
M D Harrison and D J Duke

Plenary

Industry-Academic-Government Cooperation to boost Technological Innovation and People Development in South Africa 15
Tjaart J Van Der Walt

Checklist support for ISO 9001 audits of Software Quality Management Systems 17
A J Walker

The IS Workers, they are a-changin' 29
Derek Smith

Research

Examination Timetabling 35
E Parkinson and P R Warren

Generating Compilers from Formal Semantics 43
H Venter

Efficient State-exploration 63
J. Geldenhuys

A Validation Model of the VMTP Transport Level Protocol 75
H.N. Roux and P.J.A. de Villiers

Intelligent Systems

Automated Network Management using Artificial Intelligence 87
M Watzenboeck

A framework for executing multiple computational intelligent programs using a computational network 89
H L Viktor and I Cloete

A Script-Based prototype for Dynamic Deadlock Avoidance 95
C N Blewett and G J Erwin

Parallelism: an effective Genetic Programming implementation on low-powered Mathematica workstations 107
H. Suleman and M. Hajek

Feature Extraction Preprocessors in Neural Networks for Image Recognition 113
D Moodley and V Ram

Real-Time Systems

The real-time control system model - an Holistic Approach to System Design 119
T Considine

Neural networks for process parameter identification and assisted controller tuning for control loops 127
M McLeod and VB Bajic

Reference Model for the Process Control Domain of Application 137
N Dhevcharran, A L Steenkamp and V Ram

Database Systems

The Pearl Algorithm as a method to extract infomation out of a database 145
J W Kruger

Theory meets Practice: Using Smith's Normalization in Complex Systems 151
A van der Merwe and W Labuschagne

A Comparison on Transaction Management Schemes in Multidatabase Systems 159
K Renaud and P Kotze

Education

Computer-based applications for engineering education 171
A C Hansen and P W L Lyne

Software Engineering Development Methodologies applied to Computer-Aided Instruction 179
R de Villiers and P Kotze

COBIE: A Cobol Integrated Environment 187
N Pillay

The Design and Usage of a new Southern African Information Systems Textbook 195
G J Erwin and C N Blewett

Teaching a first course in Compilers with a simple Compiler Construction Toolkit 211
G Ganchev

Teaching Turing Machines: Luxury or Necessity? 219
Y Velinov

Practice and Experience

Lessons learnt from using C++ and the Object Oriented Approach to Software Development 227
R Mazhindu-Shumba

Parallel hierarchical algorithm for identification of large-scale industrial systems 235
B Jankovic and VB Bajic

Information Technology and Organizational Issues

A cultural perspective on IT/End user relationships 243
A C Leonard

Information Security Management: The Second Generation 257
R Von Solms

Project Management in Practice 267
M le Roux

A Case-Study of Internet Publishing 271
A Morris

The Role of IT in Business Process Reengineering 285
C Blewett, J Cansfield and L Gibson

Abstracts

On Total Systems Intervention as a Systemic Framework for the Organisation of the Model Base of a Decision Support Systems Generator 299
D Petkov and O Petkova

Modular Neural Networks Subroutines for Knowledge Extraction 300
A Vahed and I Cloete

Low-Cost Medical Records System: A Model 301
O A Daini and T Seipone

A Methodology for Integrating Legacy Systems with the Client/Server Environment 302
M Redelinghuys and A L Steenkamp

Information Systems Outsourcing and Organisational Structure 303
M Hart and Kvavatzandis

The relational organisation model 304
B Laauwen

The Practical Application of a New Class of Non-Linear Smoothers for Digital Image Processing 305
E Cloete

A Technology Reference Model for Client/Server Software Development 306
R C Nienaber

The Feasibility Problem in the Simplex Algorithm 307
T G Scott, J M Hattingh and T Steyn

Author Index 309

List of Contributors

Vladimir B Bajic
Centre for Engineering Research,
Technikon Natal,
P O Box 953
Durban 4000

C N Blewett
Department of Accounting
University of Natal
King George V Avenue
Durban 4001

Justin Cansfield
Department of Accounting
University of Natal
King George V Avenue
Durban 4001

Tom Considine
Apron Services (Pty) Ltd
P O Johannesburg
International Airport
1600

Eric Cloete
School of Electrical Engineering
Cape Technikon
Box 652
Cape Town

I Cloete
Computer Science Department
University of Stellenbosch
Stellenbosch
7600

O A Daini
Department of Computer Science
University of Botswana
Gaborone
Botswana

Nirvani Devcharan
Umgeni Water
Box 9
Pietermaritzburg
3200

P J A de Villiers
Department of Computer Science
University of Stellenbosch
Stellenbosch
7700

Ruth de Villiers
Department of Computer Science and Information Systems
UNISA
Box 392, Pretoria, 0001

G J Erwin
Business Information Systems
University of Durban-Westville
Private Bag X54001
Durban 4000

G Ganchev
Computer Science Department
University of Botswana
PBag 0022
Gaberone, Botswana

J Geldenhuys
Department of Computer Science
University of Stellenbosch
Stellenbosch
7700

Louise Gibson
BIS, Dept Accounting & Finance
University of Durban
Pvt Bag X10
Dalbridge 4014

Mike Hart
Department of Information Systems
University of Cape Town
Rondebosch
7700

M. Hajek
Department of Computer Science
University of Durban-Westville
Pvt Bag X54001
Durban 4000

A C Hansen
Dept of Agricultural Engineering
University of Natal
Private Bag X01
Scottsville 3209

J M Hattingh
Department of Computer Science
Potchefstroom University for CHE
Potchefstroom 2520

Boris Jankovic
Centre for Engineering Research
Technikon Natal
P O Box 953,
Durban 4000

Paula Kotze
Department of Computer Science and
Information Systems
UNISA
Box 392
Pretoria, 0001

J W Kruger
Vista University
Soweto Campus
Box 359
Westhoven 2124

A C Leonard
Dept of Informatics
University of Pretoria
Pretoria
2000

Ben Laauwen
Laauwen and Associates
P O Box 13773
Sinoville
0129

Mari Le Roux
Information technology, development: project
leader
Telkom IT 1015
Box 2753
Pretoria 0001

P W L Lyne
Dept of Agricultural Engineering
University of Natal
Private Bag X01
Scottsville 3209

Rose Mazhindu-Shumba
Computer Science Department
University of Zimbabwe
Box MP167
Harare, Zimbabwe

Meredith McLeod
Centre for Engineering Research,
Technikon Natal,
P O Box 953
Durban 4000

D Moodley
Computer Management Systems
Box 451
Umhlanga Rocks
4320

Andrew Morris
P O Box 34200
Rhodes Gift
7707

R C Nienaber
Technikon Pretoria
Dept of Information Technology
Private Bag X680
Pretoria 0001

E Parkinson
Department of Computer Science
University of Port Elizabeth
Box 1600
Port Elizabeth 6000

Don Petkov
Department of Computer Science and
Information Systems
University of Natal
PBag x01
Scottsville 3209

Olga Petkov
Technikon Natal
Box 11078
Dorpspruit 3206
Pietermaritzburg

N Pillay
Technikon Natal
Box 11078
Dorpspruit 3206
Pietermaritzburg

V Ram
Department of Computer Science and
Information Systems
University of Natal
PBag x01
Scottsville 3209

Melinda Redelinghuys
Department of Computer Science and
Information Systems
UNISA
Box 392
Pretoria, 0001

Karen Renaud
Computer Science and Information Systems
UNISA
Box 392
Pretoria, 0001

H N Roux
Department of Computer Science
University of Stellenbosch
Stellenbosch
7700

T G Scott
Department of Computer Science
Potchefstroom University for CHE
Potchefstroom
2520

T Seipone
Department of Computer Science
University of Botswana
Gaborone
Botswana

Derek Smith
Department of Information Systems
University of Cape Town
Rondebosch
7700

Anette L Steenkamp
Department of Computer Science and
Information Systems
UNISA
Box 392
Pretoria, 0001

T Steyn
Department of Computer Science
Potchefstroom University for CHE
Potchefstroom 2520

H. Suleman
Department of Computer Science
University of Durban-Westville
Pvt Bag X54001
Durban 4000

A Vahed
Department of Computer Science
University of Western Cape
Private Bag X17
Bellville 7530

A Van der Merwe
Computer science and Informations Systems
UNISA
P O Box 392
Pretoria,0001

Tjaart J Van Der Walt
Foundation for Research and Development
Box 2600
Pretoria, 0001

K Vavatzandis
Department of Information Systems
University of Cape Town
Rondebosch
7700

Y Velinov
Dept Computer Science
University of Natal
Private Bag X01
Scottsville 3209

H Venter
Department of Computer Science
University of Port Elizabeth
Box 1600
Port Elizabeth 6000

H L Viktor
Computer Science Department
University of Stellenbosch
Stellenbosch
7600

R Von Solms
Department of Information Technology
Port Elizabeth Technikon
Private Bag X6011
Port Elizabeth 6000

A J Walker
Software Engineering Applications
Laboratory
Electrical Engineering
University of Witwatersrand
Johannesburg

P Warren
Computer Science Department
University of Natal
P/Bag X01
Scottsville 3209

Max Watzenboeck
University of Botswana
Private Bag 0022
Gaberone
Botswana

TEACHING A FIRST COURSE IN COMPILERS WITH A SIMPLE COMPILER CONSTRUCTION TOOLKIT

Dr. G. F. Ganchev
Computer Science Dept., University of Botswana
P. Bag 0022, Gaborone, Botswana
Phone: (267) 308221, E-mail: ganchev@noka.ub.bw

Abstract

We describe the use of a toolkit designed to support the Compiler Construction course in the University of Botswana. The toolkit is based upon the principles of simplicity, modularity and flexibility. Its educational goal is to maintain a balance between theoretical material and the practical presentation of concepts. We view the students as active participants in Computer Aided Learning . They actively explore and control the interactions and monitor the data and control flow in the compilers they build. The feedback provided by the toolkit's interface helps the students understand where they are in the compilation process.

Keywords

Computer Aided Learning, Compiler Construction, Scanning, Parsing, Grammar, Context Analysis, Code Generation, Stack Machine

1. Introduction

In many universities Compiler Construction is no longer a compulsory undergraduate Computer Science course. Perhaps one of the reasons is the general perception that learning compilers can be complex, while only a small percentage of graduates will be involved in compiler writing in their careers. Nevertheless the subject area "Programming Languages" of the ACM Computing Curricula [ACM-91] includes at least four main knowledge units that are closely related to understanding compilers. These knowledge units are Representation of Data Types, Sequence Control, Run-Time Storage Management, and Language Translation Systems.

Our experience indicates that the difficulty in learning compilers does not necessarily lie in the complexity of the topic itself. The cause is really two-fold: one can be characterized as a problem of foundation, the other as a problem of presentation:

- Students often lack real understanding of some fundamental concepts that are prerequisite to the topic. They are capable of obtaining good test scores, but fall apart when asked to apply their knowledge in practice. Modern compilers are syntax driven. Learning compilers requires some ease with formal languages and automata. Students should have some initial experience in the theory in order to understand the current syntax analysis methods, and their impact on the other compiler components.

- The available introductory textbooks either concentrate on one particular compilation model and even one example programming language, and then explain their model in a great detail but miss the general state of the art picture, or have an encyclopedia-like approach covering a range of methods and techniques, but lacking concrete guidelines for their integration and implementation.

As a first step in addressing these issues, in the UB curriculum we have a separate course in Languages and Automata. Still, in the Compilers course we try to rely on as simple a set of concepts as possible. We organized our course with a focus on the concepts that differentiate one compilation model from the other, and included a closed laboratory component intended to allow the students to rapidly build components of compilers for languages that they define. We encourage the students to explore and choose particular models as described below for each of their compiler components.

2. The Course Contents

The course is designed for undergraduate single major Computer Science students. Our goal was to maintain a balance between theoretical material and the practical presentation of concepts.
The students taking the course have passed a course in Formal Languages and Automata and a course in Programming Languages.

After surveying a number of textbooks, analyzing the advanced material in sources as [Hol-90], and considering the algorithms of several models, we developed our course from the themes listed below. We were convinced that a set of tools designed to support these themes would provide the students a meaningful and lasting learning experience not only in compiler construction, but also more generally in building large software systems.

Our course starts with an introduction to language translation systems encompassing the range from assemblers to compilers and interpreters with emphasis on syntax directed translation. Then we present the main parsing, translation, and code generation techniques in use today with many examples. We recommend two texts for the course - [Wat-93] and [Aho-86], however none of them is strictly followed, and the lectures contain a lot of material that binds the concepts being presented.

During the first half-semester we cover scanning (case-type and with finite automata) and parsing. Two bottom-up and two top-down parsing techniques are discussed: mixed precedence, LR(k), Recursive Descent and LL(k). All are supported by examples and exercises. The second half of the course concentrates on semantic analysis, run-time storage management (including routines and data types representation), and code generation (including expression evaluation and sequence control). The modern emphasis on compiler construction tools is underlined. The horarium is three lecture hours and one two-hour guided laboratory per week. A laboratory assistant helps the students learn the toolkit in closed laboratory sessions.

3. The UB Compiler Construction Toolkit

The University of Botswana toolkit differs from other similar systems [Ben-90,Hol-90] by the intentional stress on simpler concepts. In [Mar-95] M. Maredi and H.J. Oosthuizen point out the problems associated with the poor mathematical background of a large group of South African students. We have to overcome a similar obstacle. Many of our students experience difficulties in understanding automata based models, attribute-value flow, semantic specifications. In contrast, more mechanistic concepts are easily understood. For example, the students understand better the construction of a case-type scanner than building a finite automaton from a regular expression. They understand more easily the construction of precedence-based shift-reduce parsers than LR(k) parsers. They prefer building translation schemes to defining semantic functions. Fig. 1 shows the model of a compiler adopted in our course.

We made a fundamental decision that our tool will be used primarily by the individual students rather than by the instructor in the classroom (although it could certainly be used for classroom demonstrations). Everyone knows the importance of the student role in the learning process. It has been stressed by several authors [Cou-93,Sch-93] that one of the most important ingredients of a successful learning tool in science and engineering education is its flexibility and its ability to be adapted according to the student's needs and personal ideas. This is why our toolkit offers the students differing alternative models to probe their understanding of different parts of compilers they build. We have included the following:

- two scanner generators. One of them generates case-type scanners from a BNF description of the source language syntax. Some conventions must be observed for the generated scanner to be immediately ready for use. Its main advantage is its very simple structure which students are able to understand and modify without difficulties. In the closed laboratories this scanner generator is used in all cases which are not primarily concerned with scanning. The second generator generates scanners from a description of the lexicon by regular expressions. As mentioned above, we prefer the first, more mechanistic approach. In the closed exercises the second generator is used only to illustrate a more general and modern approach. Because of the standard interface used by both generators, in their individual work the students are able to choose the approach that they prefer.

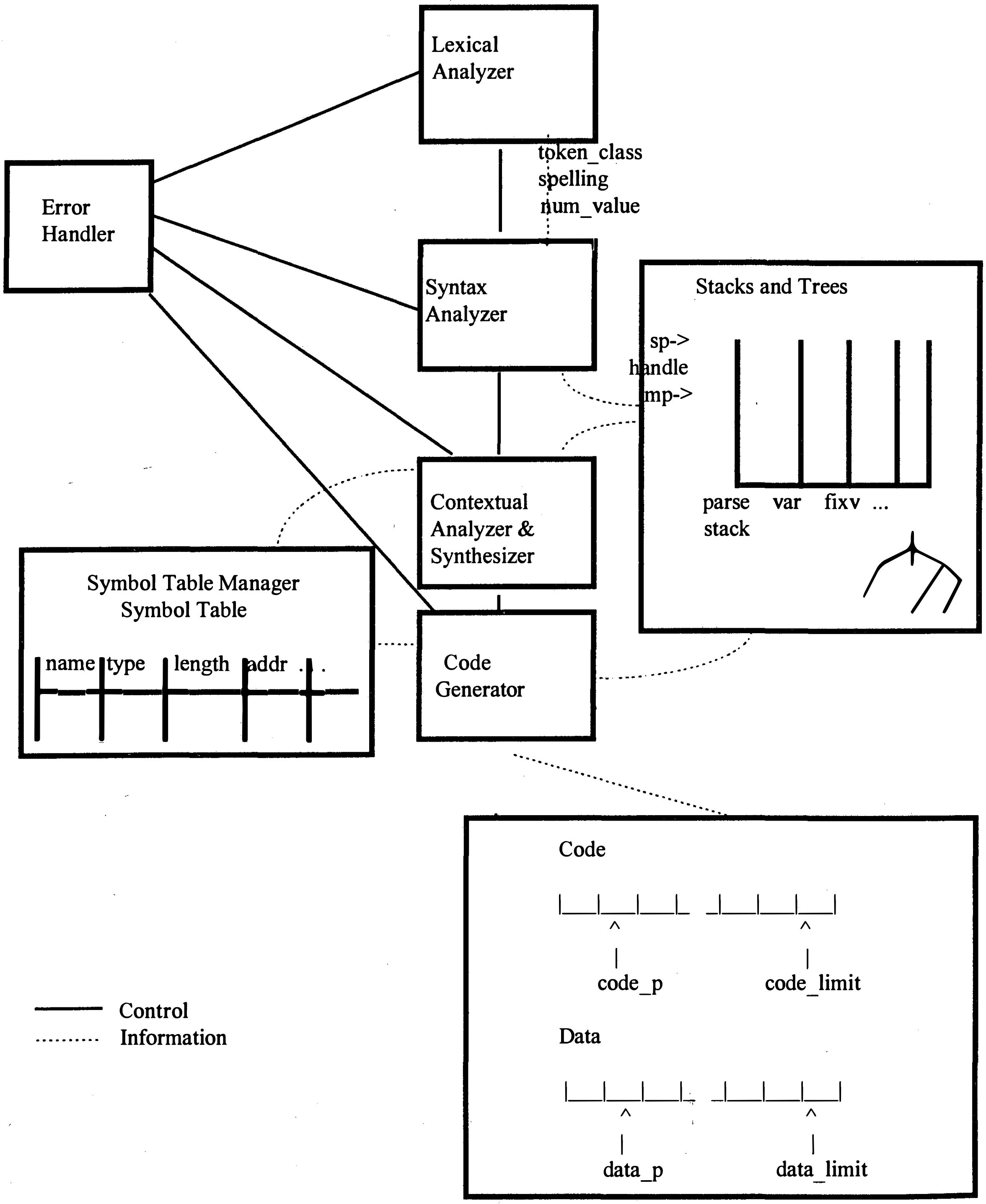


Figure 1. The adopted structure of a compiler

- two parser generators - a top-down (LL(1)) and a bottom-up (mixed precedence). Only the bottom-up parser generator was operational in the last academic year, however the students practiced top-down parsing by building hand-written recursive descent parsers from extended BNF descriptions of the source languages.

- an almost empty "skeleton" routine for semantic analysis and synthesis. The students try their semantic actions interactively before supplying the code to fill in the "skeleton" routine. The different options that they could explore originate from the different possibilities for defining the concrete syntax and linking it with the semantics of the source language.

- a set of primitive code generation routines that generate assembly code for the TAM virtual machine (see below). The students call these routines from their own semantic programs.

- an error handler, a major part of which is an adjustable syntax error recovery routine implementing "panic" mode of recovery [1]. In addition students are encouraged to implement and explore recovery by replacement and error productions [1].

- a stack-machine emulator. We have adopted the TAM virtual machine described in [9], and have developed an interactive user interface and an assembler for it.

- utility programs. At the time being these include symbol-table, stack and tree manipulating routines.

- a standard interactive user interface that allows prototyping and debugging of compilers being built by the students.

Below we shortly discuss the last three components.

3.1. Defining Symbol Tables

Understanding symbol-table manipulation is a key point in learning contextual analysis and synthesis. Our goal was to implement a reusable and easily adjustable component that would conveniently allow visualization. We chose to implement the symbol table as a heterogeneous C++ class having objects of another class as an attribute. Each symbol table entry consists of a scope level number, a name and an attribute pointer. In this way all symbol-table operations were implemented in advance, while the structure of the attribute was left open. In a minimalist scenario, the students are only expected to specify the structure of their attribute. This was the case in our closed laboratories. In order to make the exercises simpler and to provide a standard visual image of the table, we used attributes with a fixed structure, however the heterogeneous class approach allows much more flexibility. Not only can the attribute be different for different compilation exercises, not only can it be arbitrarily complex (for example, a tree structure), but also each symbol-table entry
can have a different structure for its attribute. This could be useful for more advanced cases or optimal utilization of memory.

An interesting side effect of the flexible implementation of the symbol table was the active interest of several of our students in object-oriented programming.

3.2. Defining and Manipulating Trees

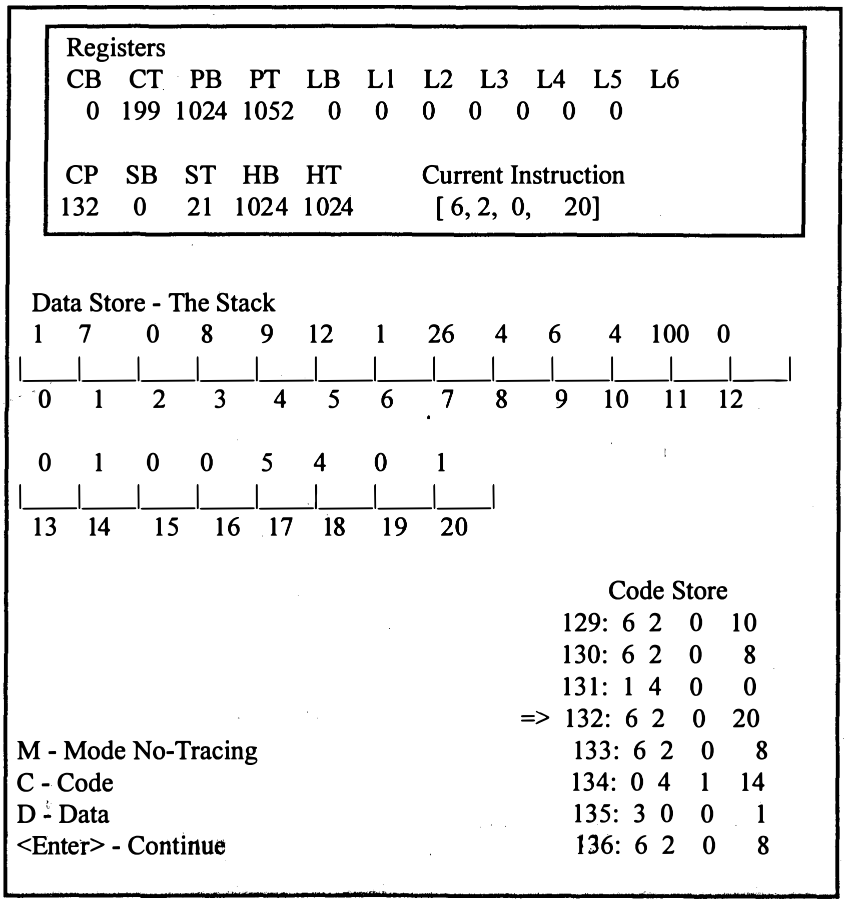
Parse trees and abstract syntax trees are popular and well understood intermediate representations, but it would be unrealistic to expect that the students will have the time to implement tree structures in the time scheduled for our course. Instead we implemented a C++ class that allows flexible tree manipulation compatible with the one described in [Wat-93]. We eliminated the limitation of [Wat-93] on the arity of nodes. For visualization we chose a two-dimensional representation of trees as more appropriate for our educational goals, instead of the widely used linear representation.

3.3. The Stack Machine

The code generation part of a compiler is very much dependent on the target machine. Different machines have different addressing conventions and register configurations. A virtual stack target machine has many advantages for a first course in compilers, like ours. Firstly, using a stack target machine eliminates the problem of intermediate storage and register allocation for expression evaluation. Secondly, the stack is the natural run-time storage organization for languages with nested scopes and recursive routines. Both texts [Wat-93] and [Aho-86] refer to virtual stack machines, but the presentation in [Wat-93] is tightly bound to such a machine. We found this presentation very useful and easily understandable by the students.

The TAM stack machine [9] has two separate stores for code and data. The data store accommodates a stack and a heap growing in opposite directions. All evaluation takes place on the stack. Primitive arithmetic, logical and other operations are treated uniformly with pre-programmed functions and procedures. An important advantage for our course is the fact that the procedure call and return conventions are implemented at the TAM machine level. This simplifies code generation, while still allowing students to observe the adopted run-time storage organization. A number of registers are dedicated to specific purposes.

Our implementation of the TAM stack machine includes a loader and an interpreter. Each of them illustrates a topic in our course. The interpreter can be run in a step-by step mode, thus allowing students to observe the execution of their compiled programs. Fig. 2 is a snapshot of a screen showing our user interface to the TAM interpreter. The current instruction is at address 132 of the code store (pointed by the register CP). The program is stored from location 0 (register CB) to location 199 (register CT). Next available location in the stack is 21 (register ST), the stack base is 0



```
Registers
CB  CT   PB   PT   LB  L1  L2  L3  L4  L5  L6
 0  199 1024 1052   0   0   0   0   0   0   0

CP   SB  ST   HB   HT      Current Instruction
132  0   21  1024 1024      [ 6, 2, 0,   20]

Data Store - The Stack
1   7   0   8   9   12  1   26  4   6   4  100  0
|___|___|___|___|___|___|___|___|___|___|___|___|___|
0   1   2   3   4   5   6   7   8   9   10  11  12

0   1   0   0   5   4   0   1
|___|___|___|___|___|___|___|___|
13  14  15  16  17  18  19  20

                                         Code Store
                                      129: 6 2  0  10
                                      130: 6 2  0   8
                                      131: 1 4  0   0
                                   => 132: 6 2  0  20
M - Mode No-Tracing                   133: 6 2  0   8
C - Code                              134: 0 4  1  14
D - Data                              135: 3 0  0   1
<Enter> - Continue                    136: 6 2  0   8
```

Fig. 2. A snapshot showing the TAM interpreter interface

(register SB). The heap is empty (HB = HT = 1024). We are executing the global part of the program (LB=0, and all link registers - L1 to L6 contain zero). The primitive routines addresses are stored from location 1024 (PB) to location 1052 (PT).

We also implemented an assembler for the TAM machine. There were several reasons for this implementation.

- Our syllabus includes coverage of assemblers as simple translation systems. The forward reference problem and its solution via backpatching are illustrated on the example of assemblers. We also use assemblers to illustrate languages with monolithic structure.

- We cover assemblers before code generation, and use the TAM assembler to introduce the TAM machine itself.

- Our code generation routines generate mnemonic TAM assembler code rather than numerical TAM machine code. This greatly increases the readability of the generated code and helps the students understand the code generation templates for control structures and expression evaluation, as well as identify the possibilities for optimization. As forward references are dealt with by the assembler, this also simplifies the contextual analysis and allows the students to concentrate separately on its other aspects - identification for nested scopes and type checking.

As a bonus, the TAM assembler and interpreter will be used for illustrations in two other courses - Machine Organization and Assembler Programming.

3.4. The Compiler User Interface

As mentioned above, our decision was to make a learning, rather than teaching toolkit. The process of learning compiler construction involves prototyping the behaviour of the compiler, as well as locating and repairing errors in compilation. The location of a defect may not be obvious from the generated target code. The user interface must allow the student to study the behaviour of the compiler in order to find the source of a defect.

Our toolkit views the student as the active and only participant in the learning process. It provides a platform from which the student can actively explore and control the interactions and monitor the data and control flow in the compiler being built. The student can interact with the various parts of the compiler, examine and change data values at any time. The feedback provided by the interface helps the student understand where he is in the compilation process. There are several paths of interaction between the toolkit and the student:

- step by step output of the canonical parse on the screen and in a listing file. This includes the production applied and the current sentential form at each step of the parse.

- screen output of the contents of all stacks. The students can change the stack values at each parse step. This is particularly useful when prototyping semantic analysis and synthesis routines which manipulate stacks parallel to the syntax stack.

- screen output of the symbol table. After examining its contents, the students can change it as appropriate.

- direct emission of code or comments in the object file. Students can enter the code to be emitted from the keyboard.

- screen output of the generated code and data. At each step of compilation the students can view and if necessary change the code and/or the data generated so far.

4. The Laboratory Component of the Course

Designing the laboratory component for a course is not that different from designing the course itself. You need to have in mind who you are teaching, what you are teaching, and how you will teach it. The stress in our course is on simpler concepts that can be mastered in the limited scheduled time. We take into account the difficulties our students have with advanced theoretical material. We try to support the lectures with numerous examples, case studies and practical work.

There are 14 closed practical exercises included in the course. Here we will mention only the most important of them. An essential assumption for the success of each laboratory session is that students know in advance their tasks and do some minimal theoretical preparation for them.

4.1. Scanning with Case-Type Scanners

This is the first scanning exercise. Its purpose is to familiarize the students with the standard scanner interface and with the structure of the case-type generated scanners. They define a language in BNF, construct a scanner for it, and interface the scanner to a main program (e.g. to output each source line in reverse order, search for given words, print the source program with indentation depending on certain tokens etc.). The students design their own main program or choose its functionality among several suggested ideas.

4.2. Canonical Parse and Synthesis

This exercise is done in the fifth week of the semester. It resembles an example done in class. The students study a very simplified compiler that still generates meaningful code. The purpose is to illustrate the syntax driven nature of the compiler and to show how using an additional stack the compiler can keep track of the run-time location of operands and intermediate results in expressions. The corresponding ideas are central in the course. They are introduced in the preceding lectures and recur and are further developed throughout the semester.

4.3. Assemblers. Simple Languages with Monolithic Block Structure. Backpatching

The TAM assembler is a second example of a simple translation system built with our toolkit. The students study its structure, follow the backpatching process, and get acquainted with the TAM abstract machine and its user interface. Sample assembler programs for searching and sorting arrays are provided, but students are expected to write their own examples.

4.4. Syntax Error Recovery with Top-Down and Bottom-Up Parsing

This is an important exercise in which the students try to limit the impact of source program syntax errors on the syntax analysis. They design sets of stop-symbols for so called "panic" error recovery and experiment with them. They verify the statement that the set of stop-symbols is very much language dependent and blindly including too many stop-symbols has an equally negative effect as including too few of them. Students are encouraged to use the second half of the scheduled time to include in the syntax of their language error productions or to experiment substituting an expected symbol for an erroneous one.

4.5. Semantic (Contextual) Analysis

This exercise illustrates the concepts of identification and type checking introduced in the lectures. A skeleton structure for the compiler is provided, as well as sample identification and type checking routines, and hints how to use them. After constructing their semantic analyzer, the students are expected to test it with correct source programs and with programs containing some typical errors, such as undeclared identifiers, double declarations in the same scope, type mismatch in an assignment, type mismatch in an arithmetic or boolean expression, missing return statement in a function body, parameter type mismatch in a function call, etc.

4.6. Run-Time Storage Organization for Languages with Nested Block Structure

Two exercises are devoted to this topic. The source language has only declarations and assignment statements with no arithmetic or other operations. The purpose is to concentrate on run-time representation of different data types, stack storage allocation, and parameter passing protocols. The students experiment with the compilation process and observe the behaviour of their compiled programs on the TAM stack machine.

4.7. Code Generation

There are two exercises for this topic. The first one uses a simple language with monolithic structure, expressions with arithmetic and logical operations, assignments, branch and loop statements. Auxiliary routines for building code templates are provided. The students are expected to incorporate these in the compiler. The second code generation exercise is the last for the course. It integrates the code generation with all the other techniques learned during the semester. The students use a simple, but full featured source language.

5. Conclusion

There are several factors of Computer Science education that make the application of our toolkit feasible. The most important one is perhaps the necessity of supporting the theoretical considerations in the concrete subject area with numerous practical examples. These make the specific knowledge more digestible for students and give them the feeling of quantitative evaluation of theoretical considerations. The other factor is the need for the students to apply the Software Engineering knowledge acquired in a course that is taught just a semester before the Compilers course. Building a complete compiler for a non-trivial language can constitute a term project for a team of students and certainly provides an exiting software engineering experience. It is worth mentioning that our toolkit itself is being implemented mainly through individual and group student projects.

We have recently obtained several tutor suggestions for extensions of our toolkit. The main ones are: a log of the student interactions with the system and a help dialogue when no interactions are made. We intend to use a weighted approach to asses the useability of the toolkit's components and decide on the further directions for improvement.

6. References

[ACM-91] Computing Curricula 1991. Report of the ACM/IEEE-CS Joint Curriculum Task Force. ACM Press, IEEE Computer Society Press 1991

[Aho-96] Aho, A.V., R. Sethi, J.D.Ullman, Compilers. Addison Wesley 1986

[Ben-90] Bennet, J.P., Introduction to Compiling Techniques, McGraw Hill 1990

[Cou-93] de Coulon, F., What Can We Really Expect from Computer Aided Learning in Engineering Education. In: Proceedings of International Conference on Computer Aided Education CAEE 93, Bucharest 1993. Ed. D. Ioan, p.3-8

[Hol-90] Holub, A. I., Compiler Design in C. Prentice Hall 1990

[Mar-95] Maredi M., H.J.Oosthuizen, A Problem Solving CAI - Factor Q. Computers and Education, Vol. 25, No 4, December 1995

[Sch-93] Scherbakov N., in oral contribution at the International Conference on Computer Based Learning in Science, CBLIS 93, Vienna, 18-21 December 1993