Checklist support for ISO 9001 audits of software quality management systems

AJ Walker
Software Engineering Applications Laboratory
Electrical Engineering, University of the Witwatersrand, Johannesburg, South Africa

Abstract: *This paper presents an overview of a national project to develop a checklist to support audits and assessments of quality management systems in the field of software in terms of the compliance requirements of the ISO 9001 international standard for quality management. This national project is reviewed in terms of: project requirements, quality objectives and quality management practices; participation by industry locally and internationally; the use of the Internet for communication and document distribution; the product development and review process; the trialling (or validation) of the product; the incorporation of guidance from the Software Engineering standards emerging from ISO\IEC\JTC1\SC7. These standards offer guidance on international good practices over a broad range of topics, and the medium and long-term benefits and impact of the development of this product.*

1 Introduction

ISO 9001 [1] has emerged as the undisputed international benchmark for quality management. Since its introduction in 1987 enterprises in more than 70 countries have established quality management systems based on the ISO 9000 family of International Standards. Tens of thousands of these organisations (more than 100 000 by end 1995) have invested in independent verification by 'registration' - or 'certification' - bodies that their quality systems conform to ISO 9000 standards. The ISO 9000 certificates then issued by the registrars to these enterprises can be used by the latter to create confidence among their clients in their ability to deliver goods and services that meet the clients requirements. The number of ISO 9000 certificates issued world-wide is expected to exceed 250 000 by the year 2000.

The ISO 9001 standard is a generic model for quality assurance in design\development, production, installation and servicing. The requirements of the standard have to be interpreted by each organisation wishing to be registered formally as evidence of meeting its requirements.

While the primary market demand for ISO 9001 has come from the manufacturing sector, the standard is now being applied to all aspects of economic activity, including software development, health care, security services, project management, and many others.

The development of software was regarded as sufficiently important for ISO to release a guidance document in 1991 providing an interpretation of the application of ISO 9001 to the development of software products and services (ISO 9000-3)[2]. This initiative was followed up in the UK by the Department of Trade and Industry who developed a sector-specific registration scheme for the software industry, known as the TickIT Scheme. Today this scheme has registered more than 900 companies in the UK alone, with strong interest in the scheme being shown in many European countries and Australia.

With the exception of the TickIT scheme [3], there is little commonality in the approach to auditing software companies against the requirements of ISO 9001. This lack of uniformity creates a serious problem for companies which seek registration to ISO 9001 in the field of software development, since there is a large gap between the ISO 9001 clauses and their interpretation for the field of software. So much so that this creates considerable problems for those who create and maintain software quality management systems against the ISO 9001 standard, and those who interpret those requirements for compliance purposes (quality systems auditors).

One of the commonly used tools in quality management is a checklist, which may contain a set of criteria against which the efficacy of a process is judged.

Remarkably there is no internationally (or nationally) agreed Checklist against which compliance of software systems to the requirements of ISO 9001 can be assessed.

In view of this a project was initiated in November 1994 by the Software Engineering Applications Laboratory (SEAL) to develop a national standard (or more accurately *Recommended Practice*) under the auspices of the South African Bureau of Standards Technical Committee for Information Technology (TC71.1) for this purpose.

This paper presents an overview of this national project in terms of:

- project requirements , quality objectives and quality management practices
- participation by industry locally and internationally,
- the use of the Internet for communication and document distribution
- the product development and review process
- the trialling (or validation) of the product
- the incorporation of guidance from the Software Engineering standards emerging from ISO\IEC\JTC1\SC7. These standards offer guidance on international good practices over a broad range of topics.
- the medium and long-term benefits and impact of the development of this product.

2 A checklist for auditing software quality management systems

At the May 1995 meeting of SABS National Committee for Information Technology (TC71.1) launched a project to develop a checklist for auditing software systems in terms of the requirements of the international standard for quality management ISO 9001. The goal of the standard is to provide a common framework for assessments and follow-up audits for companies engaged in the development of software for supply to an external customer, or where software is developed on a large scale for internal applications.

It was believed that such a tool would go a long way to reducing the all to frequent adversarial relationship between the quality assurance manager on the one part, the software developers on the second part, and the external quality system auditors on the third part. Although there is a considerable need in this area, an investigation revealed that there were no suitable checklists for this purpose available, either locally or internationally.

Current international practice is, however, moving towards the use of checklists for conducting assessments\audits of ISO 9000 quality management systems, largely to bring harmony to audit and assessment practice.

In view of this the main purpose of this project is to development a checklist to support ISO 9001 compliant audits and assessments of organisations or organisational units engaged in software development where the customer for the developed software product or service may be internal or external to that organisational unit.

It is believed that this checklist will be useful to

- internal quality system auditors and quality managers, and
- to individuals or bodies who conduct 2nd or 3rd party surveillance assessments/audits of quality management systems in the software industry.

3 Project requirements

3.1 Product requirements

The first task of the project team was to develop, review and approve the Requirement Specification for the Checklist. These requirements were categorised into those which can be considered as *functional* and *non-functional* .

Functional requirements are tangible and can be measured, while non-functional requirements can be viewed as desirable goals or objectives for the project - nice to have but impossible to measure.

The following *functional* requirements were identified:

a. The requirements of ISO 9001 (as indicated by the 'shall' in each clause) will be probed by a searching question (or series of questions) to identify the extent of compliance of the system under review.

b. The Checklist will address the domain of software.

c. The development of the Checklist questions will be strongly guided by current and emerging international good practices, and will seek to use compliance indicators being used for this purpose elsewhere.

d. The Checklist shall be available in both hardcopy and electronic format.

e. The layout of the Checklist will be guided by applicable SABS Recommended Practices i.e. ARP 013: Drafting and Presentation of Standards [4].

f. The header of the Checklist table will support the conduct of the audit\assessment, and make provision for recording:

   i. Client
   ii. Date
   iii. Reference number for the assessment/audit
   iv. Person(s) interviewed (Shown as a table, with the headings of name, initials, function).
   v. Auditor(s)\Assessor(s)
   vi. The columns of the Checklist table will support the conduct of the audit\assessment by making provision for: Auditor initials, Assessee\ Auditee initials, Checklist\ Compliance question, Company document identification, Result: [Categories A and B: Compliance; N - Non-compliance] [Categories C - F: P - Present; A - Absent] X - Not applicable, Implementation/ Observation/ Comments, Implementation Result, expressed in the [Categories A and B: Compliance; N - Non-compliance] [Categories C - F: P - Present; A - Absent] X - Not applicable, Reference to Findings Report

The following *non-functional requirements* were identified:

a. The use of the Checklist will serve to:

   i. enhance customer confidence in the client quality management system
   ii. improve the effectiveness and efficiency of audits\assessments
   iii. improve the objectivity of the assessment\audit

b. That international recognition of the product will be promoted.

3.2 Quality requirements

Beside identifying technical requirements for the product, quality objectives were identified for the process applied to the development of the Checklist and extent to which the product meet the technical requirements.

3.2.1 Project quality objectives

The quality requirements for the process applied to developing the Checklist were defined as:

a. To manage this product development in compliance with ISO 9001 requirements.

b. The Committee Draft stage will be used to apply the Checklist in practical assessment/auditing situations to determine the utility of the questions and to elicit feedback for validation purposes.

3.2.2 Product quality objectives

On the other hand, the quality objectives for the questions comprising the Checklist were required to demonstrate [5]:

a. **Objectivity**: a question is objective if it is possible to provide the answer without the opinion of the Checklist user.

b. **Completeness**: a question is complete if all the components needed to specify its meaning are present.

c. **Repeatability**: a question is repeatable if applied several times by the same Checklist user produces always the same answer.

d. **Reproducibility**: a question is reproducible if applied by different users always produces the same answer.

e. **Usefulness**: a question is useful if its answer contributes to the evaluation process.

f. **Measurability**: a question is measurable if it is possible to determine the attributes and their measures.

g. **Specific:** a question results in an attainable response.

h. All the *shall* requirements of ISO 9001 are addressed in the Checklist.

The extent to which these quality characteristics are exhibited by the Checklist questions may be evaluated by field trialling and by the application of formal inspection and review techniques.

4 Local and international collaboration

The project drew upon the following resources:

a. **Core Group Members:** (13) Individuals who are software quality system managers in local companies.

b. **Extended Core Group Members:** These include colleagues overseas experienced in software quality management (9) and a number of individuals locally who have shown interested in trialling the product in their companies (3 at the present time).

c. **Organisational Representatives of the SABS TC71.1 Information Technology Committee:** This group of individuals are responsible for approval of the developed product.

5 Project practices

5.1 Project communication

Project communication depends heavily upon the use of the Internet for communication and document distribution.

The SEAL File Server is the repository of the project documents and records.

All core group members have username and password access to the management products, technical products and records supported on the SEAL File Server.

The emerging Checklist technical products publicly available and are accessible using anonymous FTP access to the SEAL File Server thereby providing access to the Checklist products for trialling and validation. (See Appendix A).

A mail list has been created to support exchange of information and ideas between core and extended core group members. (See Appendix B)

5.2 Document management and control

The ISO 9001 Clause 4.5 document control requirements are comprehensively applied to all documents and recorded emanating from the project.

Briefly, the impact of such compliance includes:

a. Revision control is applied to all documents.

b. All documents and records are numbered and recorded in the project Master Document List.

c. Documents are 'issued' by placing them on the SEAL File Server and then issuing an e-mail 'Document Issue Notice' which advises members of the Document Name, Revision, Number and file path. Each individual is responsible for downloading the updated documents using ftp.

5.3 Configuration management

All projects artefacts are supported by a Configuration Management Plan, which describes the:

a. Document naming conventions

b. Record naming conventions

c. Description of project directory structure

d. Archiving arrangements

These conventions were defined at the start of the project and have ensured a common understanding of the project artefacts as the project has evolved.

5.4 Project tracking and control

The Project Management Plan supports details of the following:

a. Project dependencies

b. Human resources required

c. Human resources available

d. Team roles and responsibilities

e. Hardware and software required

f. Any training needs

g. The Work breakdown Structure (WBS), obligations and schedule, and log of completed tasks

Items a) - f) were assembled at the start of the project and are largely static. The WBS is actively managed as the project evolves.

5.5 Product reviews

Three types of reviews are used in the project:

a. **Project reviews:** These are undertaken in the context of formal meetings of the core group members. The purpose of the meeting is to review the status of the project and to plan in detail the next phase of product development.

   i. *Meeting Inputs*: Meeting agenda, minutes of last meeting, Project work breakdown structure - schedule and log of completed activities

   ii. *Meeting outputs:* Meeting minutes; updated Project work breakdown schedule and log

b. **Product development meetings:** These are undertaken by small taskgroups comprising 4 or 5 core group members. They are held specifically to technically elaborate the Checklist questions.

   i. *Meeting inputs:* Applicable software engineering standards, ISO 9000 series quality assurance and quality management standards; current revision Checklist questions.

   ii. *Meeting outputs:* Next revision Checklist questions.

c. **Product inspection meetings:** These are undertaken by small tasks groups comprising 4 core group members. Formal product inspection methods are used. The primary purpose of the meeting is to apply quality control measures to the Checklist questions and to provide confidence that the product technical and quality requirements are met.

   i. *Meeting inputs:* Current revision Checklist questions; forms for recording inspection and review decisions;

   ii. *Meeting outputs:* Next revision Checklist questions; quality records comprising the completed inspection and review forms.

5.6 Product development management

Developing national or international standards is unavoidably a resource intensive and time-consuming process. The goal is to achieve consensus amongst the various stakeholders on the technical attributes of a new product - which may exert a considerable impact on prevailing practices, particularly if the standard affects contractual arrangements or legislation.

In view of this management authorisation operates on a number of different levels and is governed by product progress through the Working Draft, Committee Draft and Draft SA Standard stages, which are formally governed by SABS Recommended Practice ARP 017 [6].

a. New Work Item (NWI)

The requirements specification for the proposed Checklist was assembled and submitted to SABS Information Technology (TC71.1) in June 1995. The NWI proposal was formally approved in November of the same year.

b. Working Draft Stage (WD)

The first project-level meeting was held in early December 1995 where the team concentrated on two tasks: refining the product requirements and secondly, defining the project and product quality criteria. Effort was then committed to developing the front-end support document (i.e. front page, table of contents, and supporting information), the Checklist framework, and a basic set of questions to support ISO 9001. The output of this stage was Rev 0.20 comprising 21 documents, including the Introduction and a document supporting each clause of ISO 9001.

This task was completed by the target date (1 April 1996) and the product set was released as the Working Draft for a 6 week formal review period. A Call for Review was extended to the Core, Extended Core (electronic format) and to members of TC71.1 (circulated using hard copy).

The scope of the review was defined as the structure of the Checklist, rather than the detail supplied in the Checklist questions - which would be subjected to retailed review in the CD stage.

c. Committee Draft Stage (CD)

The project is presently in this stage of development. Effort is being devoted to the elaboration of the Checklist questions taking account of relevant international standards in Software Engineering (used as a source of indicators on good industry practices) and standards dealing with quality assurance (ISO 9001) and Quality Management (largely covered by the ISO 9004 series).

The output of this stage will be Revision 0.30 of the product set with development taking place over the period June to September 1996. Formal inspection reviews will be conducted on the 20 sub-products during a 3-day workshop scheduled from 4 - 6 November 1996.

While this stage is in progress the various sub-products comprising the Checklist will be tested (trialled, or validated) against the technical requirements and quality criteria.

The product (Revision 0.3) will be formally released in late November for a project level review with electronic distribution to core and extended core members, and hard copy distribution to TC71.1 members.

Should this review elicit only minor comment the product will be updated to take account of these concerns, and then forwarded to the Draft South African Standard stage.

d. Draft SA Standard Stage (DSS)

This stage is regarded as a formality in which only changes of an editorial nature are allowed. If serious technical concerns are raised the product is returned for a further CD review.

6 Product trialling and validation

The development path taken by this project is unusual in the emphasis placed on active testing of the product at each stage of the standardisation process. Indeed, the only way in serious feedback can be elicited is by testing the subproducts in actual audit or assessment situations.

The net result of this process is to enhance confidence in the use of the Checklist.

7 Medium to long-term benefits

This project is being viewed as a potential prototype of how standardisation activities might be conducted on a wider scale in this country in future.

The following aspects of the process are novel to this project:

a. the use of a formal quality management system to support the product development at all stages of the process.

b. the exclusive use of electronic networks for exchange and distribution of documentation and for project communication. Paper based documentation is only used where the project has to interface to the wider, non-electronic world.

A major consequence of applying these support processes to the project has been to dramatically reduce the development and review cycle, making it feasible to undertake the necessary technical development and review activity in a tight timeframe of 24 months, from initial New Work Item submission to SABS, to issue of the Checklist as a National Standard.

8 Impact of emerging SE standards

The questions in the Checklist are subdivided into two key categories: those which test compliance to a standard (ISO 9001) and those which offer guidance on the implementation of ISO 9001 in the software domain.

Until very recently, there were few software engineering standards which could be regarded as international, the most influential being the US Department of Defence MIL SPEC standards, and secondly the IEEE Software Engineering standards series.

The problem with the Military standards is that they are geared to the needs of that specific sector i.e. mission critical, long lifecycle products. There are exceedingly few instances of the MIL Standards being voluntarily applied in the commercial domain - largely on account of the perceived higher development cost.

While the IEEE standards have been available for a decade or more, they have not been widely applied outside of the quasi-classified product community, and essentially unknown or at best ignored by the commercial software sector.

Both standards series suffer from the limitation that they are regarded as industry specific and controlled by a powerful stakeholder i.e. the defence community.

The software engineering standards being assembled by the Software Engineering Standards Committee (SC7) operate under the Joint Technical Committee for Information Technology (JTC1) run under the auspices of the International Organisation for Standardisation (ISO) and the International Electrotechnical Commission (IEC).

SC7 presently supports 9 active workgroups resourced by ~200 international SE technical experts from 22 countries in the following technical areas:

a. **WG2: System Software Documentation:**
Development of standards for the documentation of software systems.

b. **WG4: Tools and Environment:**
Development of standards and technical reports for tools and Computer Aided Software/System Engineering (CASE) environments.

c. **WG6: Evaluation and Metrics:**
Development of standards and technical reports for software products evaluation and metrics for software products and processes.

d. **WG7: Life Cycle Management:**
Development of standards and technical reports on Life Cycle Management.

e. **WG8: Support of Life Cycle Processes:**
Development of standards and technical reports on Life Cycle Management processes.

f. **WG9: Software Integrity:**
Preparation of standards, technical reports, and guidance documents related to software integrity at the system and system interface level. In this context, software integrity is defined as ensuring The containment of risk or confining the risk exposure in software.

g. **WG10: Software Process Assessment:**
Development of standards and guidelines covering methods, practices and application of process assessment in software product procurement, development, delivery, operation, evolution and related service support.

h. **WG11: Software Engineering Data Definition and Representation:**
Development of standards and technical reports to define the data used and produced by software engineering processes, establish representations for communication by both humans and machines, and define data interchange formats.

i. **WG12: Functional size measurements:**
To establish a set of practical standards for functional size measurement. Functional size measurement is a general term for methods of sizing software from an external viewpoint and encompasses methods such as Function Point Analysis.

The SEAL Server is the national repository of these standards in electronic format where they are in Working Draft (WD), committee (CD) or Draft International Stages (DIS). (Once a standard reaches the stage of International Standard (IS) then it can only be obtained from the ISO head office in Geneva or from the national standards body (i.e. SABS).

Standards in electronic format are also subject to copyright protection, and are only made available for review or to support standards development activity.

At the present time there are 53 standards in various stages of development from WD to IS. This high level of activity places a tremendous review burden on local technical experts. To put this issue in perspective, the flagship Standards Committee is TC 176 which dealt with the ongoing development of the ISO 9000 series. TC 176 supports a work group per standard document, while in the SC7 context each work group might have between 3 and 30 standards to manage!

9 Lessons learned

Many software developers might react to a project of this nature with a big yawn. The processes described above seem to be a far cry from the chaotic practices of the conventional software development enterprise.

The project and its context i.e. the development of national and international standards in software engineering, must be taken seriously for the following reasons:

a. South Africa is now part of the world community - and conditions affecting our capacity and capability for international trade are vitally important to the national survival and well-being.

b. Compliance to the requirements of the ISO 9001 standards is a given condition for doing business with much of the industrialised world - as many companies in this country are discovering to their cost.

c. The software engineering series of ISO \IEC\JTC1\SC7 are being used as compliance indicators when evaluating the effectiveness of quality management systems of software companies seeking ISO 9001 certification.

d. The flagship standard of the SC7 series (ISO 12207 [7]) is now a required reference point for the development of all future ISO SE standards and is exerting a strong influence on the process model under consideration for the next revision of ISO 9001, due for release in 2000.

e. There strong moves in large local corporate users of Information Technology to take software quality management seriously. Evidence of this is the number of instances which ISO 9001 requirements are now indicated in contracts from software product and service suppliers.

Tertiary education in this country, with a couple of exceptions, is largely ignorant of the pressures being brought to bear on local companies. In view of this new graduates need to be technically skilled and wellversed and experienced in key project support processes (i.e. software project management, software quality assurance, software configuration management and requirements management.)

10 Acknowledgements

The author gladly acknowledges the strong support from the core and extended group members comprising the Checklist development team, and in particular the Project Leader of SABS TC71.1 (Mr Wotjek Skowronski) for his constant support and encouragement in the development of this national standard.

11 References

[1] SABS ISO 9001 (1994) Quality Systems - Model for Quality Assurance in Design/Development, Production, Installation and Servicing

[2] SABS ISO 9000-3 (1996) (DIS) 28 June 1996 Quality Management and Quality Assurance Standards, Part 3 - Guidelines for the Application of ISO 9001 to the Development, Supply and Maintenance of Software

[3] The TickIT Guide, Appendix 1: European IT Quality System Auditors Guide, Rev 3.0, 4 September 1995, IT & Software Engineering Department , BSI Quality Assurance

[4] SABS ARP 013: 1990 Drafting and presentation of standards

[5] Evaluating evaluation instruments, F Fabbrini, N Format, M Fusani, S Gnesi, Proceedings, Software Quality Management '95, 8 - 9 November 1995, South African Society for Quality and Software Engineering Applications Laboratory.

[6] Procedures for the Technical Work in the preparation of South African Standards, ARP 017:1993, GR 14.

[7] SABS ISO 12207 (1995) Software Life-Cycle Processes

Contact Details: Professor A.J. Walker, Software Engineering Applications Laboratory, Electrical Engineering, University of the Witwatersrand, Private Bag 3, P.O. Wits, 2050, Johannesburg, South Africa. Ph: +27 11 716-5469 Mobile: +27 82 452-0933; Fax: +27 11 403-1929. E-Mail: walker@odie.ee.wits.ac.za

Appendix A: Access to the Checklist

The audit Checklist can be downloaded from the SEAL Server via FTP at the following site:

seal.ee.wits.ac.za

The files are available in the directory:

ftp/pub/iso-acl/install

The following files must be downloaded:

- **isoacl2p.exe** - Self-extracting archive of Revision 0.2+ of the Word for Windows 6.0 files for the ISO 9001 Audit Checklist for Software product.
- **instal2p.txt** - Installation instructions
- **review2p.txt** - Product review invitation form

For convenience the file ISOACL2P.EXE is also supplied in uue encoded format, as a file named ISOACL2P.UUE.

The de-archived files will occupy about 2.5 M Bytes of file space.

The file format is Word for Windows Version 6.0.

Appendix B: Subscribing to the Checklist mail list

All details concerning product updates are distributed using the SEAL Mail List Server.

To receive these notices on a regular basis and to participate in discussions with others users we ask you to subscribe to the Mail List.

This registration is performed by sending an e-mail note as follows:

E-mail address: mail-list@seal.ee.wits.ac.za

Subject: - <leave empty:

Copies to: <leave empty>

Message:

subscribe iso-acl

(No other information must appear in the body of the message).

The IS Workers, They are A-Changin'

Derek Smith
Department of Information Systems
University of Cape Town
Rondebosch

Abstract

As changes in Information Technology and organisations move into the sophisticated areas of client/server computing and open, integrated architectures, the IS professional needs to develop an increasing range of skills and abilities to produce the required systems. By synthesising the results of local research and identifying important issues from international research, the author argues that both academia and industry must react together to create a spirit of lifelong learning in IS staff. Universities offering IS degrees must look at common frameworks and IS management must put more resources into the education and training of IS professionals. Only by strongly encouraging a move to lifelong learning will companies have adequate people skills to develop tomorrow's highly complex systems.

Introduction

Come gather round people wherever you roam
And admit that the waters around you have grown
And accept it that soon, you'll be drenched to the bone
If your time to you is worth saving
Then you better start swimming or you'll sink like a stone
For the times they are a-changin'.

Bob Dylan, 1964

Bob Dylan confused, mystified and excited those of us who heard him first time around in the '60s. His music was very different from the blues, from rock 'n roll and even from folk music. Dylan decided from the first to build on the old blues of Woody Guthrie, but to develop something of his own called protest blues. Maybe the young, naive Dylan knew more about the future than we did. Whatever the reasons, we all knew that his hit song about a changing world was important. What we didn't know was that the changes would continue and would happen at a faster pace.

Over the last decade, Wetherbe and colleagues at the University of Minnesota have identified the top IS issues according to IS managers. The important IS issues have started to integrate and a recent study describes issues like architecture, integrated systems and networks as being vital to company growth. (Brancheau, Janz & Wetherbe, 1995) Staff development is also mentioned as a key issue as these are the people who will be making the other issues happen.

Assuming that the discipline and practice of Information Systems comprises the three pillars of information technology, organisations and people, it is not difficult to see that there are significant and revolutionary changes happening in business in these three areas.

Organisations are becoming global and are focusing more on core competencies. Business processes are being redesigned to handle services and products more efficiently. Middle management are being retrenched in order to "flatten" organisations and reduce operating costs. Non-core activities are being outsourced to expert service providers.

Information Technology (IT) is developing at ever increasing rates. Many organisations now identify their computer environment as network-centric and the systems which run on them as strategically important to their ongoing business survival. New hardware and software announcements from suppliers arrive daily making future planning and direction-setting a complex and bewildering exercise.

People needs and people management in IS no longer match the old paradigms. Apart from a strong move to "quality of life" and contracting/consulting work modes (Handy,1995), relatively junior staff are now empowered to make significant decisions in a distributed environment and they tend to work in multi-functional, project teams instead of the traditional functional hierarchies. Indeed, Handy argues for a new world where many people work from home and offer their services as contractors. The move to outsourcing IS services and teleworking certainly lend themselves to this trend. The IS service industry has grown remarkably in recent years. According to the Datamation 100 Survey, IS services have shown the largest growth world-wide in market share compared to any other areas in the IS industry (Brousell,1993,p23). The move to create more entrepreneurs in the IS industry is not very well researched. In a study by Smith, Boakes and Murray (1994), a university degree curriculum was developed for an IS graduate who would follow an entrepreneurial career. In this research 43 entrepreneurial skills were identified from the literature. IS entrepreneurs were asked to consider the importance of these skills. Some skill areas like communication, leadership, problem solving, change management and risk assessment were identified as requiring more emphasis whereas skills like positive thinking, creativity, ethics, business planning, scenario planning, venture evaluation and entrepreneurship theory required new courseware to be developed.

Developing Information Systems (IS) in the above sophisticated business environment is a highly complex process. Apart from the organisational and people changes mentioned above, new information technologies like desktop computing, object orientation, client/server computing and interconnecting LANs/WANs are becoming commonplace.

These issues are far removed from the "old world" of COBOL programmers working on large mainframes writing functional batch systems. However, it is argued that, apart from trying to recruit new IS staff with these new technical skills, very little is being done to change attitudes of IS staff or to identify new career ladders or to adopt new approaches to education and training. In fact, the author argues that a vain attempt is being made to develop new systems using new technologies using old management methods which must be doomed to failure.

There is evidence that some universities have identified these current requirements and have modified and enhanced curricula to ensure new graduates can make this transition (Smith,1994). Indeed, the relatively new IS'95 curriculum, a combined effort by key IS players from the ACM, ICIS, DPMA and IAIM organisations, identifies the skills , knowledge and competencies required for an IS graduate over a typical three or four year degree. These skills are very similar to those identified in a local South African study (Young, Sabor and Smith,1994). Interestingly, the IS Managers in this study identified three areas of competence - business skills, interpersonal skills and information technology skills with interpersonal skills being the most important for IS professionals. When these derived skills were presented to heads of Information Systems departments, and these academics were asked what skills were taught currently and what skills would likely be taught in 5 years time, there were many omissions. Firstly, there was a great deal of disparity amongst the current IS degree content offered at South African universities. Certainly it was difficult to detect a core syllabus or a common focus. The focus seemed to be on the technical skills with a lack of emphasis on the people and business skills. The common 4-year curriculum derived from this research provided a very exciting mix of the skill areas as shown in Table 1. The majority of the subjects are taught each year - starting with introductory concepts in the first year, leading to sophisticated practice and research in the fourth year. As each of the 42 skill/knowledge areas is taught in more detail and complexity over the four years, so the students learn to integrate the different subjects into a cohesive learning experience.

Table 1 - A 4-Year Skills Framework

Skill	Year 1	Year 2	Year 3	Year 4
Written Communication	X	X	X	X
Spoken Communication		X	X	X
Selling an Idea/Concept		X	X	X
Creativity	X	X	X	X
Dealing with Cultural Differences	X	X	X	X
Group Dynamics		X	X	X
Motivation	X	X	X	X
Leadership		X	X	X
Negotiation Skills		X	X	X
Change Management		X	X	X
Strategic Thinking		X	X	X
Problem Management	X	X	X	X
Business Relevance of IT	X	X	X	X
Business Analysis	X	X	X	X
Entrepreneurial Skills		X	X	X
Business Management	X	X	X	X
General Management	X	X	X	X
Marketing	X	X	X	
Finance Principles	X	X	X	X
Ability to Relearn		X	X	X
Package Assessment		X	X	X
JAD	X	X	X	X
Executive Information Systems	X	X	X	X
Group DSS	X	X	X	X
DSS	X	X	X	X
Project Management		X	X	X
Systems Analysis	X	X	X	X
Systems Design	X	X	X	X
Multi-media	X	X	X	X
PC Skills	X	X	X	X
Graphical User Interface	X	X	X	X
RAD			X	X
Data Management	X	X	X	X
Database Management	X	X	X	X
Programming	X	X	X	X
Systems Theory	X	X	X	X
Client/Server		X	X	X
Multiplatform Skills		X	X	X
Object Orientation		X	X	X
Telecommunications Networks	X	X	X	X
BPR and Design			X	X
Systems Architecture	X	X	X	X

The problem is not whether universities and technikons can supply IS graduates with the right skills and knowledge. Provided these academics monitor the local and international research into curricula

changes and developments, most will attempt to stay with current trends although the reaction time may be somewhat slow and curricula between universities may continue to differ because of differentiation between university faculty. However, it is vital that IS academics maintain links with each other. Bishop (1996), in a study of manpower and training in IS and computer science training, only identified nine IS departments employing 72 academics. These departments produced a mere 600 graduates in 1994. With so few staff and graduates, it is important to keep close ties between IS departments in universities. Another problem seems to be with the IS professionals themselves and IS management currently employed in the larger organisations who cannot, or will not, identify the impact of the changes happening around them and who seem reluctant to put plans in place to address these problems

The Lifetime Learning Paradigm

Handy (1995) identifies a different career structure emerging in Europe where professionals work harder and longer without the tenured jobs and regular hours expected in the last generation. He argues for a "portfolio of jobs" where work is paid not by the hour but by the product or service provided. Much of this work is provided by teleworkers who do not have to commute to the office to provide the service required. This new world of work requires a professional to understand how new organisations work, how new technology affects the organisations and how employees in these organisations can be made more effective and efficient. Understanding these new factors cannot be done through old experiences but must be continually updated with trade literature and research findings.

It has often been unfairly said (but not proven), that IS professionals read very little. And what they do read tends to be articles in the popular press and the job advertisements! Whilst the IS industry is a continually changing one, perhaps all an IS professional has time for is to deliver the systems required using the tools and the technology of the current employer.

The author argues strongly for a change in this thinking. All IS staff should be strongly encouraged to develop a philosophy of life-long learning in areas of general skills and specific technical skills - to develop and broaden as well as being skilled for specific, current technologies. The benefits to this approach have been clearly seen by the author in the IS graduates during their early career years.

The Organisation's Response

In a study by Judronich (1994), systems analysts in large organisations were asked to identify new technology skills they considered were important to their future roles in the IS industry. They were then asked to identify the levels of these skills they currently had and how these skills might change over the following three years. The general consensus was that they were hopelessly underskilled for the new information technologies that were already available in the market-place. Moreover, they did not see a move by their organisations to provide wholesale reskilling funding in the future. This concern was supported by the IS Training Managers who, despite a clear understanding of these changes in the market-place, did not anticipate any significant changes in their training budgets. This approach will do little to retain good staff and will provide a poor foundation to progress towards new technologies.

The IS Professional's Response

Many IS professionals seem to be taking the easy way out and are looking to become either SAP specialists or similarly highly-focused specialists. Whilst this approach is highly lucrative in the short term, it is easy to see that this could lead to a narrow skill set and a negation of many of the author's arguments. The knee-jerk reaction from universities who would focus on skills, for example, like SAP, Microsoft products and Netware would also lead to a long-term narrowing of the three skill sets.

Research into motivation by Couger and Smith (1994) has shown that the industry consists of people with the highest growth need of any professional groupings. The expressed need to continually achieve and grow in the job can only be satisfied by ongoing job enrichment through the acquisition of new knowledge and skills in technical, inter-personal and business areas

Conclusion

It time to start swimming. All stakeholders - academics, IS managers and IS professionals will have to change. In an industry as small as the IS industry in South Africa, I do not believe we can reinvent wheels. Academics who teach IS must start to work closer together to ensure at least a core body-of-knowledge is standardised. We must follow our colleagues in America who have started to get their act together. We must help each other to develop in similar directions.

IS Managers must realise that staff development does not merely consist of the odd skills course every year. Career planning and development must be focused on both present and future technologies and staff must be provided with considerably more opportunities to study further and to develop in broader areas of business skills, people skills and technology skills. Through extending their own company libraries and increased use of university facilities, staff must be encouraged to read more journals, books, Internet ezines and business magazines and to attend ongoing academic and commercial seminars and courses. Time must be set aside to ensure learning is an important business activity not a lunchtime browse. Staff must be encouraged to present their research and knowledge in sessions where open discussions can be held in friendly surroundings. The move to lifelong learning must be viewed as an inevitability for all IS professionals.

Bob Dylan is now in his mid-50s. Although he has modified his style dramatically over the years, his early protest song has considerable relevance to IS professionals and managers. As he says, if we don't start swimming (presumably in the right direction), then we will sink like a stone. Let's rather swim.

References

Brousell, DR (ed.) : "The Datamation 100", **Datamation**, June 15, 1993, pp12-23.

Bishop, JM : "The Status of Computing Manpower and Training in Tertiary Education in Southern African Universities 1995", **South African Computer Journal**, No 16, 1996, pp A54-A65.

Couger, D & Smith, DC: "Evaluating the Motivating Environment in South Africa Compared to the United States" **South African Computer Journal (Parts 1 & 2)**, Vols 6 & 8, 1992.

Judronich, S : "Is there a Necessity to Reskill the IS Professional", Unpublished BCom(Hons) Information Systems Technical Report, UCT, Oct 1994.

Handy, C : **Beyond Certainty**, Harvard Business Books, 1995.

Smith, DC : "The Development of an IS Curricula in South Africa", Presentation at the International Association Of Information Management Conference, Las Vegas, December 1994.

Smith, DC, Boakes, JER and Murray, AJ : "Towards the Development of a Curriculum for Entrepreneurship in the IS Service Industry", Unpublished BCom (Hons) Information Systems Empirical Research Paper, UCT, Oct 1994.

Brancheau JC, Janz BD & Wetherbe JC : "Key Issues in Information Systems Management: A Shift Towards Technology Infrastructure", **MIS Quarterly**, November, 1995.

Wozniak, S : "Reskilling the IS Professional", Unpublished BCom (Hons) Technical Report, UCT, May 1996.

Young, M, Sabor, P and Smith, DC : "A Framework for an Information Systems Degree in South Africa", Unpublished BCom (Hons) Information Systems Empirical Research Paper, UCT, Oct 1994.

EXAMINATION TIMETABLING

E. Parkinson and P.R. Warren
Department of Computer Science
University of Natal, Pietermaritzburg

Abstract

We investigate the performance of simulated annealing and tabu search, two new general problem solving heuristics, on an examination timetabling problem. These two techniques are described here and issues arising from applying them to examination timetabling are discussed. The problem we use as a test case is one that appears in Fang (1992) in which genetic algorithms were used to produce approximate solutions and experimental results were presented showing the performance of genetic algorithms on this problem. Finally, we compare tabu search, simulated annealing and genetic algorithms on the basis of experimental results and draw some conclusions.

Introduction

Timetabling examinations is an activity that takes place regularly in almost all universities and educational institutions. Ever since the advent of digital computers, the possibility of applying them to timetabling has been researched---the idea of reducing the time and manpower required to construct timetables being an attractive prospect. For an overview of this research see for example Schmidt and Ströhlein (1979) which gives an annotated bibliography of work on timetabling that appeared before 1980, de Werra (1985) and Carter (1986) which deals more specifically with examination timetabling.

In order to construct timetables that will be convenient for both students and teachers, sophisticated scheduling algorithms are needed. The timetabling problem however, like many other scheduling problems, usually takes the form of a hard combinatorial optimization problem and no fast exact algorithms exist for solving it, making it necessary to resort to heuristic methods. The types of constraints on the timetable that should be incorporated can differ greatly from one institution to the next and most algorithms are designed for either simple general exam timetable models or for very specific types of constraints and models. Few algorithms can be applied to a broad range of problems and new algorithms and heuristics have to be developed if the problem changes (de Werra 1985).

Over the last few years, three stochastic techniques have been applied to combinatorial optimization and other problems with some success: genetic algorithms (GAs), simulated annealing (SA) and tabu search (TS). These methods are designed to be robust general problem solving algorithms, that is, they perform well over a wide range of problems due to the fact that they make use of very little problem specific heuristics or information. All three techniques proceed by generating possible solutions from a solution space, trying to find a solution that minimizes (or maximizes in some cases) an objective function defined over the solution space. Usually the objective function takes the form of a penalty or cost function that indicates, for each solution in the solution space, the penalty associated with it. In cases were the solution space is too large to find an optimal solution by evaluating the objective function for all solutions, even when controlled backtracking and advanced tree pruning techniques are used, these stochastic methods can find near optimal solutions quickly by evaluating only a very small fraction of the total number of possible solutions. They can also be used to find good approximate solutions to many optimization problems provided that we can associate a cost or penalty with each solution in the search space, and therefore these methods can be used on a very wide range of problems.

The Edinburgh Problem

Fang (1992) describes an examination timetabling problem encountered at the university of Edinburgh. He also goes on to investigate the performance of several genetic algorithms using different parameters and genetic operators on this problem. We won't discuss genetic algorithms in detail here. Instead we will compare tabu search and simulated annealing with the results Fang obtained with genetic algorithms for the Edinburgh problem.

The timetabling problem in Fang (1992) is specified as an optimization problem and is therefore highly suitable to the GA, TS and SA approaches. A number of exams have to be scheduled in a number of days, where each day is divided into 4 sessions — 2 sessions each half day before and after lunch. A timetable in N days is an assignment of sessions numbered N to $4N$ to the set of exams numbered 1 to K, for K exams that have to be scheduled. For each i,$0 \leq i \leq N$-1, the sessions $4i$+1 and $4i$+2 are the first two exam sessions of day (i+1) and constitutes the first half day, while sessions $4i$+3 and $4i$+4 are the next 2 sessions of the (i+1)'th day and are in the second half day. The constraints which must be taken into account by the scheduling algorithm are described by Fang as:

STRONG CONSTRAINTS:
[1] The same student cannot take two different exams at the same time. In other words, the exams cannot clash for any student. (30)
[2] Very strongly prefer no more than 2 exams per day for a given student. (10)

WEAK CONSTRAINTS:
[3] Strongly prefer not to have exams consecutive on the same half day for the same student. (3)
[4] Prefer not to have exams consecutive on different half days of the same day for the same student. (1)

In order to model these constraints as an optimization problem, a penalty is given to each instance of a violation of a constraint in a timetable being evaluated. Given a timetable, its cost is calculated as follows: For each occurrence of a violation of one of the constraints above, the number in brackets next to it is added to the total cost of that timetable. To do this a list of courses taken by each student is examined and is used to check which constraints have been violated for each course combination taken.

Fang (1992) then compares the performance of several GAs, having different parameter settings and operators, with the timetable generated manually by human experts. Each GA is run 10 times to generate 10 different timetables. The GA that performed the best produced a six day timetable with cost 45 as the best of 10 runs, compared to the timetable generated manually by a human expert which had a cost of 101 in spite of using *seven* days. In addition, the 10 runs of the GA takes about 20-30 minutes while the human expert usually takes about an hour to construct a first draft timetable followed by consultation with the students to evaluate the timetable, and then modifications are made to the first draft.

According to Fang (1992), the GA method is clearly superior to the manual method since it is faster and constructs better timetables. The question we address next is how tabu search and simulated annealing compares with the genetic algorithm approach for this specific timetabling problem. We start by describing simulated annealing and tabu search and how we applied it to this exam timetabling problem.

Tabu Search and Simulated Annealing

Both TS and SA are based on an operations research technique known as local optimization that is often used to solve combinatorial optimization problems (Eiselt and Laporte, 1987). In order to use any of these techniques to find an approximate solution to some instance of a minimization problem, we need to define the set X of feasible solutions and the objective function $c : X \rightarrow \Re$ that we are trying to minimize. That is, we are trying to find a solution S_{best} which $c(S_{best})$ is as close to the minimum of c over X as we can find with the heuristic method in limited time. In addition, we need to define a neighbourhood structure for the set of feasible solutions. We do this by defining the function $N: X \rightarrow 2^X$ such that for each $S \in X, N(S) \subseteq X$ is the set of solutions defined to be adjacent to S, also called the neighbourhood of S. Note that we are in effect defining a directed neighbourhood graph $G = (X,E)$ where the set of vertices is the set of solutions X and the set of edges $E = \{(S,S') \mid S \in X \wedge S' \in N(S)\}$.

A local optimization algorithm starts with some initial solution $S = S_{init}$, $S_{init} \in X$ and searches $N(S)$ for a solution $S' \in N(S)$, such that $c(S') < c(S)$. Then it assigns $S = S'$ and repeats the process, terminating when a solution $S_{loc} \in X$ is reached for which $c(S_{loc}) \leq c(S'), \ \forall S' \in N(S_{loc})$. Such a solution is known as a local minimum. The fact $N(S_{loc})$ does not contain a better solution does not imply that S_{loc} is a global minimum — a solution for which c is minimized over the entire solution space X.

Both TS and SA also move through the neighbourhood graph defined on the search space X by moving from one solution to a next along the edges of the neighbourhood graph, keeping track of the best solution encountered up to that point in an attempt to find a solution with low cost. They differ, however, in the method used to select the next solution from the neighbourhood of the current one. They also differ from local optimization because they do not get trapped in local minima and do not have to terminate when a local minimum is reached.

Simulated Annealing

Simulated annealing is a process that has its roots in statistical mechanics (see Davis, 1987b). It was first proposed as a possible technique for finding near global minimum solutions to large scale optimization problems by Kirkpatrick, Gelatt and Vecchi (1983). Since then SA has been successfully applied to a number of such problems, for example the travelling salesman problem (Kirkpatrick, Gelatt and Vecchi, 1983) and graph colouring (Chams, Hertz and de Werra 1987 and Johnson, Aragon, McGeoch and Schevon, (1991).

SA is based on local optimization with one important difference: When the current solution is S and a solution $S' \in N(S)$ is randomly selected with $c(S') \leq c(S)$ that solution is immediately accepted and made the current solution. If, however, a solution $S' \in N(S)$ is generated with $c(S') > c(S)$, it has a probability $e^{-(c(S')-c(S))/T}$ of being accepted. Each selection of a solution from $N(S)$ is called a trial, irrespective of whether that solution is accepted. T in the above expression is a variable of the algorithm known as the temperature. The temperature (a term that comes from the statistical mechanics analogy) is initialized to some large value at the start of the SA process and is decreased gradually as the search progresses. This will make it increasingly less likely that solutions with higher costs than that of the current one will be accepted. It is known that, in theory, under certain conditions and when the temperature is decreased sufficiently slowly, the SA process will eventually find a global minimum for many classes of problems (see Faigle and Kern, 1991). However little is known about how to implement the SA process to find good, near optimal solutions to practical problems quickly (see Johnson, Aragon, McGeoch and Schevon, 1991).

To apply the general SA algorithm to any problem, the following parameters have to be specified: The initial temperature T_{init}, together with the temperature factor $\alpha \in (0,1)$ by which the temperature is multiplied every *rep* trials to simulate the gradual lowering of the temperature. We also have to specify the terminating condition for the search. These issues will be discussed at the end of section 3.

Tabu Search

Tabu Search is another stochastic technique based on local optimization that has had some success with combinatorial optimization problems, including graph colouring (Hertz and de Werra, 1987) and also timetabling (Hertz, 1991 and 1992). It was originally proposed by Glover for a specific application and was later generalized (see Glover 1986). It seems less popular than SA, however, and little has been published on the theoretical aspects of TS.

TS operates in a fashion similar to SA and local optimization, traversing the neighbourhood graph by moving from one solution to the next along the edges of the graph, but differs in how, at each step of this iterated process, the next solution to visit is selected from the neighbourhood of the current solution. If the current solution is S, TS generates a random sample of solutions $V^* \subseteq N(S)$ with size $|V^*| = sample\_size$, where $sample\_size$ is a parameter of the TS algorithm. If a solution v_i is found while constructing V^*, such that $c(v_i) < c(S)$, then v_i is immediately accepted as the new current solution and the process is repeated. If, however we generate the entire sample consisting of $sample\_size$ solutions without finding a better solution, the candidate in V^* with the lowest cost is accepted. This process allows TS to escape from local minima where traditional local minimization algorithms would have had to terminate.

In order to prevent TS from cycling indefinitely between a small number of solutions or vertices in the neighbourhood graph, a structure known as a tabu list is employed. When TS moves from the current solution S to a new current solution S', the reverse of the move that transforms S into S' is added to the tabu list and that move is said to be tabu. In general, the exact nature of a move in this sense depends on the application. For assignment problems a move usually consists of changing the value assigned to one of the variables of the problem. For example if we transform the current solution by only changing the value assigned to variable v_i from c_1 to c_2, then the reverse of this move is taken to be any transformation that reassigns value c_1 to variable v_1. In such a case we normally store the pair (i,c_1) in the tabu list to indicate that assigning c_1 to variable v_1 is a tabu move. Whenever a move is made its reverse is added to the tabu list and moves that are tabu are prohibited from being made under a number of conditions we will explain next. Moves do not stay tabu forever, instead only the reverses of the last *T_size* (a parameter of the TS algorithm) moves made are recorded in the tabu list and the tabu status of older moves are dropped when newer ones are added. The tabu list prevents TS from returning to a solution it has already visited during the previous *T_size* moves. However, it also makes moves to solutions not yet visited tabu when we define moves as we have done here. It could happen that a solution with a cost lower than the best solution found up to that point can be reached from the current solution, but that that solution is tabu. In order to reduce the risk of making solutions tabu that are worth investigating, Glover (1986) proposed the use of an aspiration function.

The aspiration function A initialized as $A(c) = c, \forall$ costs c when TS starts. Whenever we move from a solution S_1 with cost $c_1 = c(S_1)$ to a solution S_2 with cost $c_2 = c(S_2) < A(c_1)$, we set $A(c_1) = c_2$. The interpretation of the aspiration function is that when $A(x) = y$, then the lowest cost of a solution ever moved to from any solution with cost x is y. Thus whenever we can move from a solution S_1 to S_2 and $c(s_2) < A(c(S_1))$, we know that this transition has not been made earlier during the search. With this in mind, we can safely ignore the tabu status of a move that improves on the aspiration value of the cost of the current solution, knowing that this transition has not been made previously and that this is consistent with the goal of preventing cycling.

Applying TS and SA to the Edinburgh problem.

The above two general algorithms, SA and TS, were applied to the Edinburgh AI/CS M.Sc. 91/92 exam problem as described by Fang. This requires that the exams be scheduled in 6 days subject to the constraints given in section 2. In order to do this, a neighbourhood structure has to be defined for SA and TS, terminating conditions defined and values assigned to the parameters of the two algorithms.

The neighbourhood structure

Both SA and TS require that a solution space, neighbourhood structure and cost function for the problem be specified. The solution space, X, and cost function, c, we take to be equivalent to Fang's (1992) definition as described in section 2. The solution space is defined to consist of all assignments of the $4N$ exam sessions available to the K exams. For this particular problem, $K = 44$ and $N = 6$. The cost function we also take to be identical to that of Fang that we described in section 2. This allows us to compare the costs of solutions found by TS and SA to those found by Fang's GAs.

Unlike the GA approach, SA and TS need a neighbourhood structure imposed on the solution space. We use the same structure for both SA and TS: For any solution $S \in X$ we define $N(S)$ to consist of all assignments in X that differ from S by the session assigned to exactly one exam.

Another aspect to consider is the method used in both SA and TS to generate an initial solution. A number of possibilities are available. One is to use for an initial solution one that was found by some other heuristic algorithm (if such an algorithm was available) and the use SA or TS to attempt to improve on that. We choose the simplest method for generating initial solutions, since it was sufficient for our purposes: The initial solution was taken to be some randomly generated solution from the solution space.

The terminating condition

For both TS and SA we need to define terminating conditions for the search. In order to compare our approach with that of Fang we need some condition that will ensure that the amount of work done in that case of SA and TS is approximately equal to that done by the GAs. Ideally we want to restrict running times to be the same for all 3 methods, but this would depend on the relative speed of the system on which Fang implemented his GAs. Fortunately there is a good alternative. By far the most computationally expensive operation for all 3 these techniques is evaluating the cost of a solution by iterating through the list of students to determine for each student the penalty due to violating the constraints listed in section 2. Thus to ensure that one run of our implementations of SA and TS does approximately the same amount of work as one run of the GA that we are comparing it with, we terminate the search when 15000 solutions have been evaluated -- the same number that one run of the GA evaluates. Note that 15000 is a very small fraction of the approximately $K^{4N} \simeq 2.7 \times 10^{39}$ solutions in the search space.

Parameter tuning

Next we address the question of assigning good values to the parameters of the SA and TS algorithms. Deciding on parameter values is a tricky business. Often the only way to find parameter values is by using a combination of extensive experimentation, intuition and educated guesswork. This approach was used to find the optimal parameter settings for SA and TS too. The parameters for TS were the easiest to decide on. There are only 2: *T_size* and *sample_size*. Glover (1986) noted that the optimum value for *T_size*, the size of the tabu list, appears to be approximately 7 and that this seems to be largely problem independent. We found that smaller values for *T_size* tends to deteriorate the quality of the best solution

found in 15000 evaluations, while much larger values have very little effect in terms of the quality of solutions, but that larger tabu lists slow down the search slightly. The best value for *sample_size* was found through experimenting with a number of different values to be 50. This seems to confirm the assertion by Hertz (1992) that a good value for *T_size* when using TS for a scheduling problem is to take *T_size*= number of objects to schedule. Remember that in our problem that the number of exams to be scheduled =K=44. After some experiments with TS seemed to confirm this, we decided that *T_size*=7 and *sample_size*=50 were good parameter settings. (It is interesting to note that Fang (1992) found that with GAs the best *population size* was also about 50).

The 3 parameters for SA were slightly more difficult to decide on. The parameters we needed to find values for are: T_{init}, *a* and *rep*. The parameters *a* and *rep* together determine the rate at which the initial temperature T_{init} is reduced. Unlike TS there are no simple general guidelines for determining parameter settings for SA. After extensive experimentation, good values for these parameters were found to be: T_{init} =2, *rep*=2000 and *a*=0.8.

Results

Now we report on the performance of TS and SA. Remember that Fang's (1992) best GA found a solution with cost =45 in one of 10 runs with each run evaluating 15000 solutions. Also note that we are restricting SA and TS also to 15000 evaluations and running each algorithm 10 times with the parameter settings we indicated in the previous section.

Table 1. TS, SA and GA performance on 10 runs.

Method	#Evaluations	5000	10000	15000
SA	Best	43.0	40.0	31.0
	Avg.	49.1	47.9	46.6
TS	Best	37.0	37.0	37.0
	Avg.	58.5	56.3	55.1
GA	Best	-	-	45.0

The above table shows for SA and TS the best solution found during 10 runs after 5000, 10000 and 15000 evaluations and also the average cost of the best solution found by each of the ten runs after 5000, 10000, and 15000 evaluations. For Fang's best GA, only the best solution found after 15000 evaluations is known.

Conclusions

From table 1 we can see that both TS and SA found solutions significantly better than those reported by Fang. Even after 5000 iterations both SA and TS in one of the 10 runs of each found solutions better than that reported for the GA after 10 runs of 15000 evaluations. The best run of TS found a solution with cost 37 after only 3947 evaluations without improving it further after that. This might lead one to think that TS converges to a good solution more quickly than SA, but the gradual decrease in the average best solution found by each of the 10 runs of TS would seem to refute this when we compare it to the average values for SA.

Armed with table 1 one might be tempted to conclude that SA is better than TS and that TS is better than GAs. This would be a hasty assumption. Firstly these results apply to only one very specific problem. It is entirely possible that with slight changes to the problem the situation might be different. On the other hand it is reasonable to expect that the greater the similarity a problem has to the one we experimented with, the more relevant our results will be. There are also very many variations of GAs and it may be that other variations may perform better on this problem. The results reported on here provide no evidence of this, however.

SA and TS are obviously worth considering for timetabling problems similar to the one we examined. SA, of course, is good because it produced the best results. But in order for SA to perform well careful selection of the parameters is necessary. If we make bad choices for the parameters SA will not perform very well. In addition, the parameter values for SA tend to be rather problem dependent and we might have to reconsider parameter choices when the problem changes slightly. In contrast to this, TS would be a good method to use if we do not want to spend time optimizing parameters manually. The two parameters of TS are largely problem independent and if we choose *T_size*=7 and *sample_size*= number of exams to schedule we are likely to achieve good results.

SA and TS were also applied to the UPE timetabling problem, with similar results. The algorithms discovered timetables considerably better than the manually generated timetable. While these are only two specific cases, the fact that such good results were obtained with general problem solving algorithms should be encouraging to others who wish to apply SA and TS to exam timetabling.

References

Carter, M.W. (1986). ``A survey of practical applications of examination timetabling algorithms", Operations Research, Vol. 34, pp. 193-202.

Chams, M., Hertz, A. and Werra, D. de (1987). ``Some experiments with simulated annealing for coloring graphs", European Journal of Operational Research, Vol. 32, pp. 260- 266.

Davis, L. (ed.) (1987a). ``Genetic Algorithms and Simulated Annealing", California: Morgan Kaufmann.

Davis, L. (1987b). ``Genetic Algorithms and Simulated Annealing: An Overview", in Davis (1987a), pp. 1-11.

Davis, L. (ed.) (1991). ``Handbook of Genetic Algorithms", New York: Van Nostrand Reinhold, 1991.

Eiselt, A.E. and Laporte, G. (1987). ``Combinatorial Optimization Problems with Soft and Hard Requirements", Journal of the Operations Research Society, Vol. 38, pp. 785- 795.

Faigle, U. and Kern, W. (1991). ``Note on the Convergence of Simulated Annealing Algorithms", SIAM Journal of Control and Optimization, Vol. 29, pp. 153-159.

Fang, H. L. (1992). ``Investigating Genetic Algorithms for scheduling", MSc Dissertation, University of Edinburgh Dept. of Artificial Intelligence, Edinburgh, UK.

Glover, F. (1986). ``Future Paths for Integer Programming and Links to Artificial Intelligence", Computers and Operations Research, Vol. 13, pp. 533-549.

Goldberg, D.E. (1989). ``Genetic Algorithms in Search, Optimization \& Machine Learning", Reading: Addison Wesley, 1989.

Hertz A. and Werra, D. de (1987). ``Using Tabu Search Techniques for Graph Colouring", Computing, Vol. 39, pp. 345- 351.

Hertz, A. (1991). ``Tabu Search for Large Scale Timetabling Problems", European Journal of Operational Research, Vol. 54, pp. 39-47.

Hertz, A. (1992). ``Finding a Feasible Course Schedule using Tabu Search", Discrete Applied Mathematics, Vol. 35, pp. 255- 270.

Johnson, D. (1990). ``Timetabling University Examinations", Journal of the Operations Research Society, Vol. 41, pp. 39-47.

Johnson, S.J., Aragon, C.R., McGeoch, L.A. and Schevon C. (1991). ``Optimization by Simulated Annealing: An Experimental Evaluation; Part II, Graph Colouring and Number Partitioning", Operations Research, Vol. 39, pp. 378-406.

Kirkpatrick, S., Gelatt, C.D., and Vecchi, M.P. (1983).``Optimization by Simulated Annealing", Science, Vol. 220, pp. 671-680.

Robertson, G. (1987) ``Parallel Implementation of Genetic Algorithms in a Classifier System", in Davis (1987a), pp. 129-140.

Schmidt, G. and Ströhlein, T. (1979). ``Timetable construction - an annotated bibliography", The Computer Journal, Vol. 23, pp. 307-316.

Werra, D. de (1985). ``An introduction to timetabling", European Journal of Operational Research, Vol. 19, pp. 151- 162.

GENERATING COMPILERS FROM FORMAL SEMANTICS

Herman Venter
Department of Computer Science
University of Port Elizabeth, P O Box 1600, Port Elizabeth, 6000
e-mail: csabhv@upe.ac.za

Abstract

Given a complete, formal description of the semantics of a programming language, it should be possible to generate a compiler for the language using an automated process. This is highly desirable, since it reduces the work required to produce a compiler to the absolute minimum and makes it more likely that the compiler will be correct. This paper describes a set of tools and a methodology that does just this, for a realistic class of programming languages.

Introduction

Given a complete, formal description of the semantics of a programming language, it should be possible to generate a compiler for the language using an automated process. This is highly desirable, since it reduces the work required to produce a compiler to the absolute minimum and makes it more likely that the compiler will be correct. This paper describes a set of tools and a methodology that does just this, for a realistic class of programming languages.

There are already numerous methods[Bowen,1996] for giving formal descriptions of programming language semantics. However, none of the compiler generators built around them has yet reached the point where realistic compilers can be generated for real languages.

There are also numerous other, more realistic tool kits[Langmack,1996] that aid compiler construction, but none of these can generate a compiler given only a formal description of the semantics of a programming language. In general, the work required to generate a compiler using such a tool kits is considerably more than the absolute minimum.

Perhaps the main reason why it is proving so difficult to use formal methods to generate compilers, is that the complexity of formal descriptions of programming languages usually rival or exceed the complexity of hand-crafted compilers. Moreover, these descriptions are in notations which are complex and usually very different from programming languages.

The chief contribution of the work reported here is that it provides a way to express the formal semantics of a programming language in another programming language, which makes it possible to use the techniques of object-oriented programming to manage the complexity of the description.

The key to understanding how this can be done is to realize that an abstract syntax tree can be viewed as an expression in an object oriented programming language and that evaluating this expression amounts to running the program corresponding to the abstract syntax tree.

Interpreting Abstract Syntax Trees

Given an Extended Backus Naur Format (EBNF) description of the syntax of a language, it is relatively easy to generate code which will build parse trees from source programs. With a bit of additional annotation in the EBNF, an Abstract Syntax Tree (AST) builder can easily be generated.

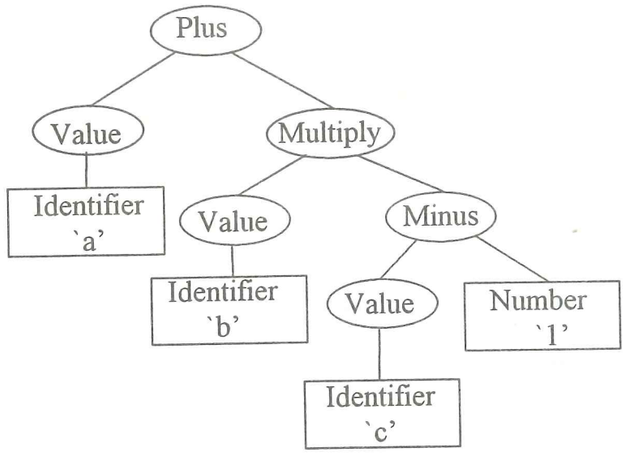
For example, given the following description:

```
expression = term {`+' term <plus 2> |  `-' term <minus 2>}
term = factor {`*' factor <multiply 2> | `/' factor <divide 2>}
factor = number | identifier <value 1> | `(' expression `)'
```

along with some additional information about numbers and identifiers, it is easy to generate a program that can turn an expression such as

```
a + b * (c - 1)
```

into the following abstract syntax tree:



It is equally easy to translate this tree into code like the following:

```
plus(value(identifier(`a')),
     multiply(value(identifier(`b')),
              minus(value(identifier(`c')), number(`1'))))
```

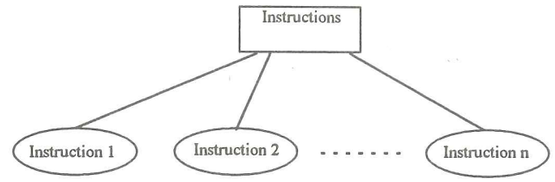
This code in turn, can easily be compiled and run, provided that suitable definitions exist for the function calls. Clearly, these functions can be specified in a programming language and constitute a formal definition of the semantics of the original language.

For example, modules Identifier and Number (written in the Slim[Venter,1996a] programming language, see the appendices) fully specify the semantics of the expression language defined by the EBNF above. To obtain an executable version of the expression program, simply compile the textual version of the AST and link in the compiled versions of identifier and number. Details of how an actual compiler for the expression language defined above, can be constructed, appear in another section.

Expressions are particularly easy to handle. Basic control flow, such as instruction sequences, if-then-elses, and loops are almost as easy. Disguised gotos like loop exits and early returns from functions and procedures are somewhat less easy, while explicit gotos and exceptions are not easy (and perhaps better left out of programming languages).

Declarations and definitions can be treated as expressions which create objects such as constants, types, variables, functions and procedures and bind them to names in the current scope.

Instruction sequences



An abstract syntax tree (AST) such as the above can be generated by customizing the tree building process. The EBNF grammar looks as follows:

```
instructions = <start_instructions> {instruction <add_instruction>}
```

When the parser reaches the first annotation, it calls routine Build_start_instruction, which creates a node labeled `instructions' and pushes it onto the parse stack. Next an instruction is parsed and a corresponding AST ends up on the parse stack. The parser then reaches the <add_instruction> annotation which causes routine Build_add_instruction to be called. This pops the instruction AST off the stack and adds it as a child of the node created by Build_start_instructions. When the end of a sequence of instructions is reached, the parse stack contains an AST such as the one depicted above. Build_add_instruction can take further actions such as adding line number information to the AST.

When this tree is finally walked, the routine Walk_instructions is called with a list of ASTs corresponding to the sequence of instructions. This list can then be translated into a string representing the instruction sequence in the target language, or it can be directly interpreted, or it can be partially evaluated.

Walk_instructions becomes a bit more complicated when the language includes gotos. This is discussed in the sections on loop exits, function/procedure returns and unbridled gotos.

If then else

```
if_instruction =
  `if' expression `then'
     instructions <lazy>
  `else'
     instructions <lazy> <if_then_else 3>
  `end' `if'
```

If-then-else constructs are handled much as they are in SmallTalk: The nested instructions are packaged into block objects which `lazily' carry out the instructions when their value methods are invoked.

In other words, one adds a Boolean class to the semantic definition and define an if-then-else method for Boolean objects, passing lazy objects corresponding to the nested instruction sequences.

The else construct can be made optional by introducing an if_then method:

```
if_instruction =
  `if' expression `then'
     instructions <lazy>
  (`else'
     instructions <lazy> <if_then_else 3> | <if_then 2>)
  `end' `if'
```

If-then-elsif-else constructs are easily transformed into nested if-then-else constructs:

```
if_instruction =
  `if' expression `then'
     instructions <lazy>
   (elsif | else | <if_then 2>)
   `end' `if'
elsif =
  `elsif' expression `then'
     instruction <lazy>
  (elsif | else | <if_then 2>) <lazy> <if_then_else 3>
else =
  `else'
      instructions <lazy> <if_then_else 3>
```

Loops

```
while_loop =
  `while' expression <lazy> `do'
     instructions <lazy>
  `end' <while_loop 2>
```

While loops are easily handled by packaging the control expression into a lazy object and adding a while_loop method to the lazy class. Repeat-until and Repeat-forever loops can be handled similarly.

For loops can be handled by introducing an iterator class and adding a for_loop method:

```
for_loop =
  `for' identifier `:=' expression `to' expression `do' <new_iterator 3>
     instructions <lazy> <for_loop 2>
  `end'
```

Loop exits

```
loop_exit = `exit' <exit>
```

Consider the while_loop method of the lazy class:

```
func while_loop(self, loop_body : Lazy) -> Entity
  result := Undefined
  repeat
    cond := value(self)
    exit when cond /= boolean.True
    result := value(loop_body)
  loop
```

To exit early from this loop, the evaluation of the loop body must return immediately when the exit instruction is executed and it must set some sort of flag that the while_loop method can check. For example:


```
func while_loop(self, loop_body : Lazy) -> Entity
  result := Undefined
  repeat
    cond := value(self)
    exit when cond /= boolean.True
    result := value(loop_body)
  until global.Must_exit_flag = True
  global.Must_exit_flag := False
```

When the loop body is translated, it suffices to translate an exit instruction into an assignment to the global flag, followed by a return instruction. If it is interpreted, the routine executing the sequence of instructions should check the flag after every instruction.

Multi-level exits can be modeled by using a global counter instead of a global flag.

Return from procedure/function

```
return_from_procedure = `return' <return>
return_from_function = `return' [expression <assign_to_result>] <return>
```

Returning early from a call to a procedure or function is similar to an exit from a loop: The return instruction is translated into an assignment to a global flag, followed by a return instruction in the target language. If the procedure/function body is interpreted, the Walk_instructions routine must check the flag after every instruction.

Early returns from functions usually involve an expression specifying the result of the call. This can be implemented as an assignment to a result variable accessible to the method implementing the function call. The easiest way to handle this is to rewrite the AST so that a return instruction with return value expression ends up as an assignment instruction followed by a return instruction.

Gotos

Goto instructions are quite problematical, since the target instruction could be in an instruction sequence which is not even being executed. Furthermore, any number of procedure/function activations may have to be terminated.

By far the best solution seems to remove goto instructions from the language being implemented.

However, if goto instructions *must* be implemented, it can be handled as follows (assuming that language rules subject labels to normal scope rules, with nested instruction sequences treated as nested scopes and therefore not visible).

Case 1, the goto instruction targets another instruction in the same sequence: This goto can be translated into a goto of the target language. If interpreted, a flag can be set with the name of the target label. Walk_instructions would have to check this flag after every instruction and select the next instruction appropriately. Something similar to a hardware instruction counter can be used.

Case 2: the goto instruction targets an instruction in an outer instruction sequence: This goto can be handled similarly to a multi-level exit, while proceeding as in Case 1, once the appropriate scope has been reached. Quite a lot of coordination must be provided in the methods modeling loops and function/procedure calls.

If it must be possible to target labels in nested scopes, quite a bit more work must be done and a lot depends on the actual semantics of the language. For example, what happens when you jump into the middle of a for loop?

Since one is modeling the semantics of the language, rather than specifying its translation into another language, it is inevitable that constructs with complicated semantics will be hard to implement using the methods outlined in this text. This is not necessarily a disadvantage.

An example specification

The source listings of the complete specification of a compiler for a small expression language appear in Appendix 1. Of the files listed there, only expr.bnf, identifier.slm and number.slm are specific to the expression language. (expr.slm is a trivial adaptation of the standard main module). It thus takes a grand total of 46 lines of code to generate an entire compiler, albeit for a very simple language.

The modules which are not program generated are discussed below.

Further examples of language specifications can be found in [Venter, 1996b]

EBNF for the expression language

```
goal = <start_list> {expression <display 1> <add_to_list>}
```

This production sets the goal for the parser. It specifies that an expression program is made up of zero or more expressions. The parser will construct an Abstract Syntax Tree (AST) for each such expression and leave a list of these on the parse stack.

The production can be read as follows: <start_list> calls a the routine Build_start_list in custbld.slm. This creates a node, labeled list, with no children and pushes it onto the parse stack. Next, an expression is parsed, resulting in a corresponding AST to be pushed onto the stack. This is popped and made a child of a node labeled display. The resulting AST is pushed back onto the stack. Then, the routine Build_add_to_list in custbld.slm is called, which pops the node labeled display, as well as the node labeled list, off the stack, makes the display node the last child of the list node and pushes the modified list node onto the stack.

Thus, when there are no more expressions to parse, the stack contains a list of ASTs. These are then traversed by the Walk function in walker.slm.

```
expression = term {`+' term <plus 2> |  `-' term <minus 2>}
```

This production will cause the parser to parse a term and push a corresponding AST onto the parse stack. Then, if the next token is a `+' or a `-' it will parse another term and push the corresponding AST onto the stack. The two ASTs are then popped off the stack, made the children of a node labeled plus or minus, respectively, and the new AST pushed onto the stack. In other words, the stack is reduced. Note that annotations in <> brackets are handled automatically when the number of items to be reduced are specified. If not, a routine which must be supplied by the language designer in module custbld.slm is called.

```
term = factor {`*' factor <multiply 2> | `/' factor <divide 2>}
```

This production is similar to the one for expression. Note that it causes multiplication and division to have a higher precedence than addition and subtraction.

```
factor = number | identifier <value 1> | `(' expression `)'
```

Note that identifiers are automatically evaluated when they appear in expressions. If we had assignments, the identifier on the left-hand side would not be evaluated.

```
identifier = (Up_let | Low_let) {Up_let | Low_let | Digit | `_' }
number = Digit{Digit}[`.'Digit{Digit}][`e'[`-']Digit{Digit}]
```

These productions specify how identifiers and numbers are to be scanned. Although scanner generators traditionally use regular expressions to specify token syntax, EBNF can just as easily be used. Note that Up_let, Low_let, and Digit are predefined terminal symbols that correspond to character classes in the scanner.

```
Tokens = identifier | number
```

This production tells the compiler generator that identifier and number are productions to be processed by the scanner generator and that they must be treated as tokens by the parser generator.


```
Ignorable = ` ' | Eol
```

This production tells the parser that the tokens corresponding to blanks and end-of-line characters should be ignored. Comments could be added to the syntax by specifying a suitable production and listing it as token as well as ignorable.

Identifier

This module represents the semantics of identifiers in the expression language.

```
import number
```

Needed to obtain access to class Number and constructor new_number.

```
export Identifier, new_identifier
```

Exports class Identifier and constructor new_identifier for use by other modules. In this case, the only module to use Identifier would be the main module of an expression program (when translated into Slim).

```
Identifier : set of Entity := {}
```

This sets up Identifier as a set of objects. Initially, the set is empty. It is populated with objects as new Identifier objects are created. In other words, Identifier always represents the current instances of class Identifier. Class Identifier is not explicitly declared, but is treated as one and the same as the set of instances.

```
name : map from Identifier to String
```

Name is an associative array that can be indexed with values of type Identifier. Thus name(x) can be set to a string value which represents the name of identifier x. In other words, this declaration sets up name as a string valued attribute of class Identifier. Name need not be exported since every object of class Identifier carries the entire name space of this module with it.

```
stored_value : map from String to Number
```

Every leaf node in the Abstract Syntax Tree (AST) corresponding to an identifier will instantiate a new identifier object when called. In other words, when expression a+a is evaluated, two identifier objects will be created. Since both of them should evaluate to the same value, the value of identifier is not stored as an attribute of the identifier object, but in a separate associative array.

```
func new_identifier(n : String) -> Identifier
  result := new(Identifier)
  name(result) := n
```

This constructs an instance of class Identifier and initializes the only attribute. Following a call to new(Identifier) the new instance is automatically added to Identifier, the set of instances of class Identifier.

```
func value(self : Identifier) -> Number
  result := stored_value(name(self))
  if result = Undefined
    repeat
      put name(self)`?' & get v
    until v in number !number is the built-in numeric type
    result := new_number(mkstr(v))
    stored_value(name(self)) := result
  end
```

The EBNF syntax (expr.bnf) ensures that this method is called whenever an identifier appears in an expression. An expression such as a + 1 translates into

```
plus(value(new_identifier(`a')), new_number(`1'))
```

The method looks up the value of the identifier, if known. If the value is not known, it obtains it interactively from the user and stores it for future look ups. In other words, for an expression such as

a+a, only the first call to value(new_identifier(`a')) will cause an input operation. Note that different instances of class Identifier will access the same storage location if they have the same name.

Number

This module represents the semantics of numbers in the expression language.

```
export Number, new_number
```

Exports class Number and constructor new_number for use by other modules.

```
Number : set of Entity := {}
```

This sets up Number as a set of objects. Initially, the set is empty. It is populated with objects as new Number objects are created. In other words, Number always represents the current instances of class Number. Class Number is not explicitly declared, but is treated as one and the same as the set of instances.

```
slim_value : map from Number to number
```

Slim_value is an associative array that can be indexed with values of type Number. Thus slim_value(x) can be set to a numeric value which represents the numeric value of number object x. In other words, this declaration sets up slim_value as a numeric valued attribute of class Number. Note that case is significant and that number is the built-in numeric type in Slim, whereas Number is a programmer defined abstract type (class).

```
func new_number(v : String) -> Number
  result := new(Number)
  slim_value(result) := mknum(v)
```

This constructs an instance of class Number and initializes the only attribute. Following a call to new(Number) the new instance is automatically added to Number, the set of instances of class Number.

```
func display(self : Number) -> Number
  put slim_value(self)
  return self
```

This method displays the numeric value of the Number object on the console.

```
func divide(self : Number, other : Number) -> Number
  result := new(Number)
  slim_value(result) := slim_value(self) / slim_value(other)

func minus(self : Number, other : Number) -> Number
  result := new(Number)
  slim_value(result) := slim_value(self) - slim_value(other)

func multiply(self : Number, other : Number) -> Number
  result := new(Number)
  slim_value(result) := slim_value(self) * slim_value(other)

func plus(self : Number, other : Number) -> Number
  result := new(Number)
  slim_value(result) := slim_value(self) + slim_value(other)
```

These methods construct and return new Number objects with numeric values equal to the results of the respective operations carried out on the numeric values of the operand objects.

Routines for customizing AST building (custbld.slm)

```
import builder walker
```


Each node in the Abstract Syntax Tree (AST) is labeled with a routine from walker.slm (and optionally, custwalk.slm) which is used to process the node during traversal (walking) of the tree. Module walker is imported here to incorporate all of its exported routines into the name space of the current module. Module builder is imported to gain access to the parse stack.

```
export Build_start_list, Build_add_to_list
```

These routines are exported for use by the parser, which will call them when the corresponding <start_list> and <add_to_list> annotations in the EBNF is reached.

```
proc Build_start_list
  Stack +:= [[Walk_list]]
```

This routine is called whenever <start_list> is reached in the EBNF. It creates a list node with no children and labels it with the Walk_list routine. When the list is walked, routine Walk_list will be called with the list of children as its parameter.

```
proc Build_add_to_list
  O1, Stack := delete_last(Stack)
  O2, Stack := delete_last(Stack)
  Stack +:= [O2 + [O1]]
```

This routine is called whenever <add_to_list> is reached in the EBNF. It pops the top two elements of the stack and appends the top most element to the end of the other. The modified element is then pushed back onto the stack. The intended use is to add AST children to a list node created by Build_start_list. See the Walk_list routine in walker.slm.

Routines for customizing AST walking (custwalk.slm)

```
export nothing
```

This is an empty module. It is used only when there is a need to customize the tree walking process. This is not often necessary, since it is typically sufficient to customize the tree building process. Customized tree walking is used only when it is necessary to utilize information gathered during the tree building process. In other words, it adds a second pass to the compiler, if needed.

Routines for AST walking (walker.slm)

These routines traverse (walk) the Abstract Syntax Tree (AST) created by the parser. In this case, a string corresponding to the translation of the AST into an equivalent Slim program is returned by the top most call (Walk) made in expr.slm. Other walker modules might directly interpret the AST, or partially evaluate it, or translate it into a different target language.

```
import custwalk
```

The routines in custwalk.slm, if any, will appear as annotations of nodes in the AST (the first elements of the lists corresponding to the nodes) and will be called by Walk.

```
export Object_id_for, Runcall, Walk, Walk_list
```

These routines are exported for use by the parser and AST builder, as well as custbld.slm.

```
func Walk(T) -> Entity
  if T(1) in {Runcall, Walk_list}
    return T(1)(tl(T))
  else
    return eval(T(1), tl(T))
  end
```

This routine is a general routine that calls specific routines to walk the AST represented by T. The first element of T is the annotation of the root node and represents the specific routine (belonging either to this module, or to custwalk.slm) that is to be used to walk the node. If the specific routine is

Runcall or Walk_list, the children of the node (the remaining elements of T) is passed as a single list. If it is any other routine, the children are passed as individual parameters.

In other words, if T = [Walk_list x y z] then the call is Walk_list([x y z]) whereas if T = [Object_id_for `integer' `123'] the call is Object_id_for(`integer', `123').

```
func Runcall(lst) -> Entity
  return lst(1) + as_string(Walk_list(rest(lst)))
```

The first parameter of this call is the name of the Slim routine to be called at run-time. In the case of the expression language this will be a routine belonging to identifier.slm or number.slm. The subsequent parameters are ASTs which represent expressions which form the run-time parameters of the call. These ASTs are walked (turned into strings) and collected into a parameter list by as_string. For example:

```
Runcall([`plus' [Object_id_for `number' `1'] [Object_id_for `number' 2]])
```

will result in the string `plus(new_number(`1'), new_number(`2'))'

```
func Walk_list(lst) -> list of Entity
  return [all Walk(i) for i in lst]
```

This function takes a list of ASTs as parameter and returns a list of walked (translated) ASTs as result.

```
func Object_id_for(cn, iv) -> Entity
  return `new_'+cn+`(\`'+iv+`\')
```

This function returns a string that represents a run-time call to a constructor of an object of class cn. The constructor is assumed to take a single string as a parameter.

```
func as_string(lst) -> String
  return `()' when lst = []
  result := `('
  for e in all_but_last(lst)
    result +:= mkstr(e)+`, '
  loop
  result +:= mkstr(last(lst))+`)'
```

This function takes a list of strings and turns it into a comma delimited, bracketed string. For example, [`x' `y' `z'] becomes `(x, y, z)'.

Optimizing Compilers

Compilers produced with the techniques outlined in this text are about as far away from optimizing compilers as one can get. Consider the output produced by the example compiler for the expression

```
a + a
```

namely

```
plus(value(new_identifier(`a')), value(new_identifier(`a')))
```

To evaluate this expression, it is necessary to construct two identifier objects and three number objects. When translated simplistically into machine code, several thousand instructions may be executed. Most of these instructions accomplish at run time, what compilers normally do at compile time.

Ideally, this should not be a problem. The output from the expression compiler must still be submitted to the Slim compiler before it becomes machine code. If the Slim compiler is really clever, it should be able to remove all the redundant instructions.

Unfortunately, this is not yet the case. In fact, using Slim as a target language is still a disadvantage, since Slim itself is implemented using the techniques described in this paper. One is thus using a really slow interpreter to interpret another really slow interpreter. (Fortunately, today's computers are so fast that using this scheme is actually quite tolerable.)

Until such time as advances in the Slim compiler makes it unnecessary to do so, there are a number of things that can be done to speed things up. The easiest is to translate the ASTs into C++ (or similar) instead of into Slim. This is fairly trivial to do. The main disadvantage is that one then has to write the semantics in C++ rather than in Slim. For simple languages this should not be too much of a pain. This is pretty much the approach currently used to compile Slim itself. The result is quite acceptable on a fast processor and can serve to produce a very useful prototype.

Another approach is to partially evaluate the AST before translating it into code. To do this, one augments the semantic classes with classes for unknown (or partially known) values. For example, to turn the expression compiler into a partial evaluator, one first turns it into a proper interpreter by changing the AST walker to directly call the semantic class methods, instead of producing code which calls them. Next, one changes the value method of the identifier class to produce unknown numbers instead of doing an actual input operation and then producing a known number. In other words, we change

```
func value(self : Identifier) -> Number
  result := stored_value(name(self))
  if result = Undefined
    repeat
      put name(self)`?' & get v
    until v in number !number is the built-in numeric type
    result := new_number(mkstr(v))
    stored_value(name(self)) := result
  end
```

to

```
func value(self : Identifier) -> Unumber
  result := stored_value(name(self))
  if result = Undefined
    result := new_unumber(number, `get')
    code_name(result) := name(self)
    stored_value(name(self)) := result
  end
```

Now, when an expression such as a+a is processed, the plus method for unknown numbers is called, instead of the plus method for known numbers. The result is another unknown number. Unknown numbers keep track of how they are to be computed.

Along with changing input operations to produce unknown numbers, it is necessary to change output operations to produce code. For example, one changes the display method of the number class from

```
func display(self : Number) -> Number
  put slim_value(self)
  return self
```

to

```
func display(self : Number) -> Number
  Code_string +:= `put '+mkstr(slim_value(self))+`\n'
  return self
```

More interesting code generation happens when one calls the display method for an unknown number:

```
func display(self : Unumber) -> Unumber
  Code_string +:= code_for(self)+`put '+code_name(self)+`\n'
  return self
```

For this to work, each unknown number must keep track of whether it was the result of an input operation or an arithmetic operation. In the latter case, it must also keep track of the operands. The code_for method looks as follows:

```
func code_for(self : Unumber) -> String
  code_for(self) := `' !only execute this function once
  return `put \`'+code_name(self)+`?\' &\n' +
    `repeat get '+code_name(self)+`: Integer until '+code_name(self)+
      ` /= Undefined'+`\n'
    when operator(self) = `get'
  result := code_for(operand1(self)) + code_for(operand2(self))
  result +:= code_name(self)+` := '
  if operator(self) = `divide'
    result +:= code_name(operand1(self))+`/'+code_name(operand2(self))+`\n'
  elsif operator(self) = `minus'
    result +:= code_name(operand1(self))+`-'+code_name(operand2(self))+`\n'
  elsif operator(self) = `multiply'
    result +:= code_name(operand1(self))+ *'+code_name(operand2(self))+`\n'
  elsif operator(self) = `plus'
    result +:= code_name(operand1(self))+`+'+code_name(operand2(self))+`\n'
  else
    assert False
  end
```

Note that the code string could easily be three address intermediate code which is then submitted to a classical code generator. Also note that, since (at most) one of the operands of an unknown number could be a known number, it is necessary to extend the class definition of known numbers with methods code_for and code_name.

An esoteric feature of the code_for method is that it produces code only the first time it is invoked. For example, the code for the expression a+a is:

```
put `a?' &
repeat get a: Integer until a /= Undefined
temp1 := a + a
```

The second call to value(new_identifier(`a')) produces the same unknown number as the first. The code_for method is thus called twice for the same object. The first time code is returned, the second time only an empty string.

Note that classes identifier and integer are now used at compile-time rather than a run-time. They come into play when the AST is walked. The result of the walk is a partial number, which in effect is another AST. The recursive calls to code_for walk the second AST, producing the desired output.

With the addition of the class for unknown numbers, the semantics of the expression language now approach the complexity of a traditional compiler. The complete partial evaluator-based compiler for expressions is given in Appendix 2.

Things become somewhat more complicated when control flow is added, but the overall complexity of compilers specified in this way still seems manageable and probably compares favourably with many conventional optimizing compilers.

Conclusion

The main idea presented in this text is that the semantics of a programming language can be readily expressed in a programming language. The idea is fairly obvious and has been around in various guises for almost as long as programming languages. As usual, however, the devil is in the details.

The method outlined in this text has evolved over a period of eight years. It has been applied to various small language subsets[Venter,1996b] as well as to one fairly complex language (Slim[Venter, 1996a]). It seems easiest to use on functional languages - languages with unrestricted gotos and pointers (like C) are perhaps more difficult to handle with this method than with translational semantics.

At the moment, compilers constructed with this method essentially amount to executable specifications. I would thus recommend the method for use in rapid prototyping of new languages and for constructing language standards.

The use of partial evaluation as a means to turn an executable specification into a production compiler still has to be explored further. Work is underway to turn the Slim semantics into a full scale partial evaluator. Preliminary results appear promising., but the bulk of the work still has to be done.

If the Slim compiler can be made clever enough, other compilers can ride on its back and the rather immodest claim in the introduction, that the work required to produce a compiler has been reduced to the absolute minimum, may well be justified.

References

[Bowen, 1996] Bowen, J. P. Formal Methods. http://www.comlab.ox.ac.uk/archive/formal-methods.html

[Langmack,1996] Langmack, O. Catalog of Compiler Construction Products. http://www.cs.upe.ac.za/staff/csabhv/papers/compiler-generator/tools.txt

[Venter,1996a] Venter, B. H. Slim. http://www.cs.upe.ac.za/slim

[Venter,1996b] Venter, B. H. Programming Language Semantics. http://www.cs.upe.ac.za/staff/csabhv/pl-semantics

An on-line version of this paper is available as http://www.cs.upe.ac.za/staff/csabhv/papers/compiler-generator

Appendix 1

expr.bnf

```
goal = <start_list> {expression <display 1> <add_to_list>}

expression = term {`+' term <plus 2> |  `-' term <minus 2>}
term = factor {`*' factor <multiply 2> | `/' factor <divide 2>}
factor = number | identifier <value 1> | `(' expression `)'

identifier = (Up_let | Low_let) {Up_let | Low_let | Digit | `_' }
number = Digit{Digit}[`.'Digit{Digit}][`e'[`-']Digit{Digit}]

Tokens = identifier | number
Ignorable = ` ' | Eol
```