The South African Institute of Computer Scientists and Information Technologists

Proceedings

of the

1996 National Research and Development Conference

Industry meets Academia

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

Edited by Vevek Ram
FOREWORD

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

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

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

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

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

Organising Chair
Dr Don Petkov (UNP)

Secretariat
Mrs Jenny Wilson

Program Chair
Prof Vevek Ram (UNP)

Program Committee
Prof Peter Wentworth, Rhodes
Dr Milan Hajek, UDW
Prof Derek Smith, UCT
Prof Anthony Krzesinski, Stellenbosch
Dr Don Petkov, UNP
Mr Rob Dempster, UNP
Prof Peter Warren, UNP
### Table of Contents

Foreword i  
Organising Committee ii  
List of Contributors vi

**Keynote Speaker**

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

**Plenary**

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

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

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

**Research**

*Examination Timetabling*  
E Parkinson and P R Warren  

*Generating Compilers from Formal Semantics*  
H Venter  

*Efficient State-exploration*  
J. Geldenhuys  

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

**Intelligent Systems**

*Automated Network Management using Artificial Intelligence*  
M Watzenboeck  

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

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

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

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

...
Real-Time Systems

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

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

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

Database Systems

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

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

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

Education

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

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

COBIE: A Cobol Integrated Environment
N Pillay

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

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

Teaching Turing Machines: Luxury or Necessity?
Y Velinov

Practice and Experience

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

Parallel hierarchical algorithm for identification of large-scale industrial systems
B Jankovic and VB Bajic
Information Technology and Organizational Issues

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

Information Security Management: The Second Generation
R Von Solms

Project Management in Practice
M le Roux

A Case-Study of Internet Publishing
A Morris

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

Abstracts

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

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

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

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

Information Systems Outsourcing and Organisational Structure
M Hart and Kvavatzandis

The relational organisation model
B Laauwen

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

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

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

Author Index


List of Contributors

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

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

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

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

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

I Cloete  
Computer Science Department University of Stellenbosch Stellenbosch 7600

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

Nirvani Devcharan  
Umgeni Water Box 9 Pietermaritzburg 3200

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

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

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

G Ganchev  
Computer Science Department University of Botswana P Bag 0022 Gaborone, Botswana

J Geldenhuys  
Department of Computer Science University of Stellenbosch Stellenbosch 7700

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

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

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

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

J M Hattingh  
Department of Computer Science Potchefstroom University for CHE Potchefstroom 2520
Boris Jankovic  
Centre for Engineering Research  
Technikon Natal  
P O Box 953,  
Durban 4000  

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

J W Kruger  
Vista University  
Soweto Campus  
Box 359  
Westhoven 2124  

A C Leonard  
Dept of Informatics  
University of Pretoria  
Pretoria  
2000  

Ben Laauwen  
Laauwen and Associates  
P O Box 13773  
Sinoville  
0129  

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

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

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

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

D Moodley  
Computer Management Systems  
Box 451  
Umhlanga Rocks  
4320  

Andrew Morris  
P O Box 34200  
Rhodes Gift  
7707  

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

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

Don Petkov  
Department of Computer Science and  
Information Systems  
University of Natal  
P Bag X01  
Scottsville 3209  

Olga Petkov  
Technikon Natal  
Box 11078  
Dorpspruit 3206  
Pietermaritzburg  

N Pillay  
Technikon Natal  
Box 11078  
Dorpspruit 3206  
Pietermaritzburg
VRam
Department of Computer Science and Information Systems
University of Natal
PBAg X01
Scottsville 3209

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

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

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

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

T Seipone
Department of Computer Science
University of Botswana
Gaborone
Botswana

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

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

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

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

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

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

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

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

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

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

H L Viktor
Computer Science Department
University of Stellenbosch
Stellenbosch
7600

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

\[ a + b \times (c - 1) \]

into the following abstract syntax tree:

![Abstract Syntax Tree](image)

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:

\[
\text{instructions} = \langle\text{start\_instructions}\rangle \{\langle\text{instruction}\rangle \langle\text{add\_instruction}\rangle\}
\]

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 \langle add\_instruction\rangle 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

\[
\text{if\_instruction} = \\
\quad \langle\text{if}\rangle \ \langle\text{expression}\rangle \ \langle\text{then}\rangle \\
\quad \langle\text{instructions}\rangle \ <\langle\text{lazy}\rangle> \\
\quad \langle\text{else}\rangle \\
\quad \langle\text{instructions}\rangle \ <\langle\text{lazy}\rangle> \ <\langle\text{if\_then\_else\ 3}\rangle> \\
\quad \langle\text{end}\rangle \ <\langle\text{if}\rangle
\]

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**

```plaintext
    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

\[
\text{goal} = \langle \text{start_list} \rangle \{ \langle \text{expression} \rangle \langle \text{display} \rangle \langle \text{add_to_list} \rangle \}
\]

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: \langle \text{start_list} \rangle calls 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.

\[
\text{expression} = \langle \text{term} \rangle \{ \langle \text{plus} \rangle \langle \text{term} \rangle \langle \text{minus} \rangle \langle \text{term} \rangle \}
\]

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.

\[
\text{term} = \langle \text{factor} \rangle \{ \langle \text{multiply} \rangle \langle \text{factor} \rangle \langle \text{divide} \rangle \langle \text{factor} \rangle \}
\]

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.

\[
\text{factor} = \langle \text{number} \rangle \langle \text{identifier} \rangle \langle \text{value} \rangle \langle \text{expression} \rangle
\]

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.

\[
\text{identifier} = (\text{Up_let} \mid \text{Low_let}) (\text{Up_let} \mid \text{Low_let} \mid \text{Digit} \langle \text{\_} \rangle)
\]

\[
\text{number} = \text{Digit} (\text{Digit}) [\langle \text{.} \rangle \langle \text{Digit} \rangle [\langle \text{e} \rangle [\langle \text{-} \rangle \langle \text{Digit} \rangle [\langle \text{.} \rangle \langle \text{Digit} \rangle]]]
\]

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.

\[
\text{Tokens} = \langle \text{identifier} \rangle \langle \text{number} \rangle
\]

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.

```plaintext
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.

```plaintext
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.

```plaintext
func value(self : Identifier) -> Number
result := stored_value(name(self))
if result = Undefined
  repeat
    put name(self) '?\n' & 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 \(a + l\) translates into

```plaintext
plus(value(new_identifier('a')), new_number('l'))
```

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.

```plaintext
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.

```plaintext
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.

```plaintext
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)**

```plaintext
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.

```plaintext
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.

```plaintext
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.

```plaintext
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').

```go
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:

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

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

```go
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.

```go
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.

```go
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

```go
a + a
```

namely

```go
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

```plaintext
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

```plaintext
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

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

to

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

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

```plaintext
func display(self : Unumber) -> Unumber
    Code_string := code_for(self) + 'put ' + code_name(self) + '
    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)+'?\" &
  'repeat get '+code_name(self)+': Integer until '+code_name(self)+
  '/= Undefined'+\"\nwhen 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))+`\nelsif operator(self) = 'minus'
  result +:= code_name(operand1(self))+`- '+code_name(operand2(self))+`\nelsif operator(self) = 'multiply'
  result +:= code_name(operand1(self))+`* '+code_name(operand2(self))+`\nelsif operator(self) = 'plus'
  result +:= code_name(operand1(self))+`+ '+code_name(operand2(self))+`\nelse
  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
  templ := 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


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
custbld.slm

import builder walker
export Build_start_list, Build_add_to_list

proc Build_start_list
    Stack +:= [[Walk_list]]

proc Build_add_to_list
    01, Stack := delete_last(Stack)
    02, Stack := delete_last(Stack)
    Stack +:= [02 + [01]]

custwalk.slm

export nothing

walker.slm

import custwalk
export Object_id_for, Runcall, Walk, Walk_list

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

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

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

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

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

Saicsit '96
**identifier.slm**

import number
export Identifier, new_identifier

Identifier : set of Entity := {}
name : map from Identifier to String

stored_value : map from String to Number

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

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

**number.slm**

export Number, new_number

Number : set of Entity := {}
slim_value : map from Number to number !number is the built-in numeric type

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

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

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)
expr.slm

import opsys parser builder walker

Parse(Command_line(2)+' .expr' 'r')
lines := Walk(Stack(1))

Close(Stdout)
ok := Assign_file(Stdout, Command_line(2)+' .slm', 'w')
Terminate_process(Myself, 255) when not ok
put 'import identifier, number'
put '
for 1 in lines
  put 'junk := '1
loop

Close(Stdout)
ok := Assign_file(Stdout, Command_line(2)+' .lnk', 'w')
Terminate_process(Myself, 255) when not ok
put '(['Command_line(2)'][identifier number]'

Close(Stdout)

Execute_command('slim '+Command_line(2))
Delete_file(Command_line(2)+' .slm')
Delete_file(Command_line(2)+'.obj')
Delete_file(Command_line(2)+'.lnk')
Delete_file(Command_line(2)+'.rsp')

Appendix 2

code.slm

export Code_string

Code_string : String := ''

expr.slm

import opsys parser builder walker code

Parse(Command_line(2)+' .expr' 'r')
junk := Walk(Stack(1))
put Code_string
identifier.slm

import number unumber
export Identifier, new_identifier

Identifier : set of Entity := {} 
name : map from Identifier to String 
stored_value : map from String to Number+Unumber

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

func value(self : Identifier) -> Entity 
result := stored_value(name(self)) 
if result = Undefined 
result := new_unumber('get') 
code_name(result) := name(self) 
stored_value(name(self)) := result 
end
number.slm

import code unumber
export Number, new_number

Number : set of Entity := {}
  slim_value : map from Number to number !number is the built-in numeric type
  func new_number(v : String) -> Number
    result := new(Number)
    slim_value(result) := mknum(v)
  func code_for(self : Number) -> String
    return ''
  func code_name(self : Number) -> String
    return mkstr(slim_value(self))
  func display(self : Number) -> Number
    Code_string += `put ' + mkstr(slim_value(self)) + \n''
    return self
  func divide(self : Number, other : Entity) -> Entity
    return unumber.divide(self, other) when other in Unumber
    result := new(Number)
    slim_value(result) := slim_value(self) / slim_value(other)
  func minus(self : Number, other : Entity) -> Entity
    return unumber.minus(self, other) when other in Unumber
    result := new(Number)
    slim_value(result) := slim_value(self) - slim_value(other)
  func multiply(self : Number, other : Entity) -> Entity
    return unumber.minus(self, other) when other in Unumber
    result := new(Number)
    slim_value(result) := slim_value(self) * slim_value(other)
  func plus(self : Number, other : Entity) -> Entity
    return unumber.minus(self, other) when other in Unumber
    result := new(Number)
    slim_value(result) := slim_value(self) + slim_value(other)
import code
export Unumber, new_unumber

Unumber : set of Entity := {}
operator : map from Unumber to String
operand1 : map from Unumber to Entity
operand2 : map from Unumber to Entity
code_name : map from Unumber to String

calc new_unumber(op : String, opnd1, opnd2 : Entity) -> Unumber
result := new(Unumber)
operator(result) := op
operand1(result) := opnd1
operand2(result) := opnd2
code_name(result) := new_code_name()

calc 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
display

Name_counter : Integer := 0

calc new_code_name -> String
  Name_counter += 1
  return 'temp'+'mkstr(Name_counter

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

calc divide(self : Entity, other : Entity) -> Unumber
  return new_unumber('divide', self, other)

calc minus(self : Entity, other : Entity) -> Unumber
  return new_unumber('minus', self, other)

calc multiply(self : Entity, other : Entity) -> Unumber
  return new_unumber('multiply', self, other)

calc plus(self : Entity, other : Entity) -> Unumber
  return new_unumber('plus', self, other)