An Implementation of the Haskell Language

Diomidis Spinellis

June 1990

Abstract

This report describes the design and implementation of HASKELL system. The areas implemented are the lexical analysis, parsing, intepretation of the lambda tree, and machine code generation. Because of the size, complexity and novelty of the language many of these areas present particular difficulty. A considarable amount of meta-programming was used in order to tackle the size of the project.

Contents

List of Tables

Introduction

> Haskell is a general-purpose, purely functional programming language exhibiting many of the recent innovations in functional (as well as other) programming language research, including higher order functions, lazy evaluation, static polymorphic typing, user-defined datatypes, pattern matching and list comprehensions. It is also a very complete language in that is has a module facility, a well-defined functional I/O system, and a rich set of primitive data types including lists, arrays, arbitrary and fixed precision integers, and floating point numbers. In this sense HASKELL represents both the culmination and solidification of many years of research on functional languages. [Hud89, p. 381]

In this report I present the implementation of the front and back ends (scanning, parsing, interpreting and generating machine-specific code) for a HASKELL system.

This is the first implementation of the language done in the imperative paradigm that I am aware of [1] so considerable scope for experimentation with its novel features existed.

The very large scale of the project (more than 12000 lines of code) was an additional challenge in organising it. Simplifying the implementation, minimising errors and increasing efficiency, were acomplished by designing, implementing and using four small meta-languages and a number of code management tools.

The novel features of this—committee designed—language such as layout, orthogonality between operators and function identifiers, operators of variable

[1] The two other implementations that are under-way (Glasgow and Yale) are being written entirely in HASKELL [Hud89, p. 405].

precedence and associativity, source layout rules, type classes and modules have never existed in a single language before [2]. For some of the problems created by the interaction of the various features (such as parsing curried function applications) extended bibliographic research revealed that no formal solutions existed and thus solutions had to be formalised.

The focus of attention on the front end of the language was efficiency, reliability and implementation of the full HASKELL standard. All of these goals have, in my opinion, been achieved. On the back end the emphasis was more on experimentation with code generation, profiling and debugging.

[2] PL/I might qualify if presented appropriately.

Chapter 1

Lexical Analysis

The lexical analyser reads Haskell programs and converts them into a set of tokens. It keeps track of the current row and column numbers, interprets string and character escapes removes comments and other white space, converts integers and floating point numbers and implements parts of the layout rule that are applicable during the lexical analysis phase.

1.1 Technical Overview

The implementation and theoretical issues of lexical analysers are described in [ASU85, pp. 83 – 157].

In [ASU85, p. 89] *lex* three approaches to the implementation of a lexical analyser are listed. They are ordered, from easiest and least efficient and most complicated and efficient as follows:

1. Using a lexical-analyser generator.
2. Writing a lexical analyser in a conventional systems–programming language using the I/O facilities of the language for input.
3. Writing the lexical analyser in assembly and explicitly managing the reading of input.

The solution finally adopted was a hybrid of 2 and 3; a hand crafted analyser in C using a special input library.

The advantages of using language development tools are presented in [JL87]. A number of tools which automate the task of creating lexical analysers such as

lex [Les75], *mkscan* [HL87], *flex* [Pax89] and the one described in [Heu86] are available.

One of the tools described, the lexical analyser generator *lex* [Les75] was initially used. *Lex* is not widely adopted. In [AKW79, section 5] the authors indicate that they are using *lex*, but other implementation descriptions such as [Joh82, p. 6] outline hand crafted analysers. This trend was verified by examination of the source code of publicly available language systems such as the GNU C compiler [Sta89] and PERL [Wal88]. The reason for this appears to be that analysers generated by *lex* are slow and consume a lot of space. In fact, [Heu86] suggests that nearly all lexical analysers for production compilers are written by hand.

A formal design methodology for writing scanners is discussed in [DGM80]. A practical design outline for a hand crafted parser that is roughly the base for my design is found in [Wai86]. The major difference of my implementation is that his `mkint`, and `mkfpt` routines are implemented with inline code.

[Heu86] gives the basic rules to follow for creating a fast scanner:

- "touch" characters as few times as possible,
- avoid procedure calls.

Both of these rules have been followed in the design of the scanner.

A most efficient method for expanding tabs is described by [Wai85]. He proposes to minimise the number of operations on input, eliminating the column counting variable. Instead the column number is dynamically computed using the pointer to the input buffer and another variable containing the amount of extra space inserted. The other variable needs only updating when a tab is encountered. The method was not chosen because of its additional complexity and the frequent number of times the column number is examined in HASKELL .

The C standard input-output library initially used is described in [KR82, Appendix A]. An alternative, more efficient approach, can be found in [Hum88]. The *fast input library* described minimises the copying of data and allows traversing of the input using a character pointer. Speed advantages are mainly gained in line oriented applications. Since most HASKELL tokens are not allowed to cross a line boundary its use is very appropriate in this context. It was implemented in C for the UNIX and MS-DOS operating systems.

A list of tasks which complicate the design of an analyser is given in [Joh82, p. 8]. In HASKELL this is complicated by the existence of the layout rule. In [ASU85, p. 84] the possibility of simplifying the lexical analysis by splitting it into two

phases, “scanning” and “lexical analysis proper” is outlined. I decided for reasons of efficiency to use the integrated approach also found in [Joh82, p. 9].

In many parts of the code a sentinel is used to speed up a loop. This idea was used in the lexical analyser of the SAIL compiler and can be found in [Ben82, p. 55]. The character classification tables described in section 1.5.1 are patterned on the *ctype* macros [V7.82, ctype(3)]. The use of such tables to speed up processing can be found in [Joh82, p. 8].

In section 1.4 an alternative design for the lexical analyser is presented.

1.2 General Description

The top routine of the analyser is `yylex`. This routine every time called returns the numeric code of a new token. All the tokens that can exist are automatically numbered with numbers above 255 by the parser generator *yacc* [Joh75]. Token values below 256 are used to indicate single ASCII characters. Tokens that have additional data associated with them (such as the strings, integers and symbols) have that data or a pointer to it copied into the appropriate field of the union variable `yylval`. This is also defined by the parser generator.

`Yylex` is used as an interface to a ring buffer. The ring buffer contains a list of tokens that need to be returned to the parser. Access routines allow the addition of tokens to both ends of the buffer, for reasons outlined in sections 2.3.5 and 2.6. When the buffer is empty then the routine `gettoken` is called to supply the next token.

`Gettoken` is the lexical analyser engine. Initially implemented using *lex* it was reimplemented in C using a switch case for every possible input character. Most tokens are recognised by a set of macros and returned immediately. More complicated cases such as comments, strings and characters are dealt by the routines `process_comment`, `process_string` and `process_character`.

At a lower level the `input` macro returns the next character from the input stream. It counts the line and column number, virtually expanding tab characters. It also provides for a single character pushback needed by most recognition loops.

The interface to the operating system is provided by an implementation of the *fio* library ([Hum88] [V885, fio(3)]). The main routine used `Frdline` returns a character pointer to a beginning of a line. According to [Hum90] after opening a file no other initialisation is required for using the library.

1.3 The Layout Rule

The layout rule of HASKELL allows for the layout of the problem to convey information normally represented by the use of braces for block specifiers and semicolons for declaration separators. With the layout rule if an item is indented more than the rest in a place where an open brace is expected then an implicit brace is inserted. Items that are indented by the same amount have a semicolon inserted before them and a return to a previous level of indentation closes an implicitly opened brace.

The September specification of HASKELL [HWA+89, p. 3] defined that an opening brace could be expected at three different places:

1. after the export list of a module,
2. after the keyword `where` and
3. after the keyword `of`

Since the first place is determined by the grammar the lexical analyser could not by itself determine when an opening brace was required. A tie-in with the parser was needed. This was done by the addition of a rule `open_brace` in the grammar which before asking for a token from the lexical analyser called the function `brace_needed`. That would then check to see if a brace was available in the file. If one was not available it would push one in the input stream.

The April version of the document [HWA+90, p. 3] simplified this situation by requiring a `where` identifier after the export list. Thus the lexical analyser can always know when an open brace can possibly be inserted.

The rule for inserting an open brace specifies that the next token needs to be examined. This can be implemented by having the scanner recursively call itself, so that whitespace and comments are removed, tabs are expanded and columns counted.

This approach was found to be very messy. In normal operation the scanner is called by the parser. Some state information such as the line and column numbers needs to be saved between calls. The recursive call destroyed the state information and made this approach very difficult to implement correctly. An alternative approach was opted for. A *mini-scanner* was written. The *mini-scanner* recognises only whitespace-class tokens, like comments, and genuine whitespace. Whenever the next token needs to be examined the *mini-scanner* is called.

Indentation levels are kept in an integer stack. A stack pointer always points to the current indentation level. The stack is of finite length and checks against over and underflow are made. Indentation forced the addition of one more piece

of state information to the scanner. In certain places a suitably indented token may terminate more than one levels of blocks. Consequently the scanner needs to return a number of closing braces. A static variable is checked every time the scanner operates to see if some more closing braces need to be returned.

The April HASKELL document gives an additional rule for the layout. Specifically a semicolon or brace is inserted whenever the next syntactical item is illegal, but a semicolon or brace would be legal. This rule can not be handled by the lexical analyser. Is section 2.6 I describe how this problem was solved by modifying the parser.

1.4 Alternative Design

From the above it should be clear that the implementation of the scanner is not very clean. The operation of the scanner is dependent on seven state items. These are:

1. The current column.
2. The current line number.
3. A pointer to the next character to be processes in the input stream.
4. A possible pushback character, its line number and column. This is used to implement a one level pushback stack.
5. The column of the last character processed. This is needed because if a character is pushed back the column can not be just decrement by one due to complications with tab expanding.
6. All the indentation levels up to the current indentation.
7. Possible additional close braces that may need to be returned.

As is described in section 2.5 a further buffer is needed to push arbitrary tokens.

I tried to think of a programming paradigm that could naturally accommodate the needs of the scanner in a clean way. The best approach would be using *coroutines* [Hoa78]. An example on how programs can be structured using coroutines can be found in [Pal82].

The scanner and the parser work as coroutines. Whenever the parser needs a token, it is suspended and the scanner is run. The scanner works until it is ready to pass a token back to the parser and then gets suspended. In this way the state information is implicitly kept by the coroutining mechanism resulting in a cleaner

implementation. One further advantage of this approach is that it leads to easy parallelisation.

Two ways of implementing the scanner as coroutines were examined. The one involves breaking the program into two processes. The one process is the scanner and the other the parser. The two processes communicate using a pipe and are suspended and resumed under operating system control.

The other way is implementing coroutines inside C . A monitor is used to control the coroutining. When the monitor starts a C function as a coroutine it allocates a new stack in dynamic memory and assigns that stack to the function. Communication is also done via the monitor, which can switch processes by switching the stacks. An implementation of coroutines for C can be found in [Bai85a].

None of the two approaches was implemented because of various problems.

- The two process approach is very inefficient. Every token needs to pass through the operating system two times (one write and one read) in order to go from one process to the other. This involves two system calls and a context switch. A typical time for a system call is $350\mu s$ [Fed84, p. 1796] and for a context switch $600\mu s$ [Fed84, p. 1798]. The total overhead for a single token would be $1.3ms$. The performance finally obtained (see section 1.5.6 was $128\mu s$ per token) so this approach would make the scanner ten times slower.

- Both approaches are not portable. There is no standard C library for coroutines and switching the stack — although possible within C — needs to be done in a different way on different machine architectures and compiler argument passing conventions. The two process implementation would limit the system to the Unix operating system.

- The operation of the scanner and the parser needs closer coupling than that provided by a pipe. They both need to share the symbol table as the lexical category of some tokens can be redefined by the user.

1.5 Coding for Speed

The lexical analyser was initially built using the *lex* [Les75] lexical analyser generator. Some aspects of HASKELL required however modifications to the code produced by *lex*. This meant adding another level of control around the input processing of *lex*, slowing down the whole operation. This leaded to the scanning process being redesigned with the aim of improving speed. The areas where

improvement could be made were token recognition, character copying, memory allocation and symbol table maintenance.

1.5.1 Token Recognition

The top level of token recognition is built using a C `switch` statement. A different case is used for almost every character. Since the table is quite dense it is compiled by most C compilers into an indexed jump table. When a character arrives, it is used as an index into a table of code locations and a jump is made to that location. The separate entries then process the token according to its lexical category. Separate functions are used to scan characters, strings and comments. All other scans are done in the main function in order to avoid the function calling overhead. Macros are used instead of functions for the same reason.

A procedure was developed for writing macros in a disciplined rather than the usual *ad-hoc* manner in order to enhance code readability and maintainability. Specifically macros were written in a multi line form like normal procedures using indentation to make them more clear. The C "alternative" operator `? :` was indented like an if statement and brackets were used instead of braces. Finally macros that contained semicolons in them were wrapped in a `do { macro } while(0)` to give them the syntactic quality of a single statement (otherwise they might trigger obscure bugs if used as a single statement after an `if`.

In a number of places tests were needed to check if a character belonged to a certain class (octal digit, identifier, symbol etc.). There is a C set of macros, the *ctype* macros, that are designed to distinguish between character classes, but many character classes defined by HASKELL , such as octal digits and operator symbols are not part of the *ctype* macros. The usual way to handle such situations is to code as `strchr("class", ch)`, the intent being to check if `ch` is one of the characters in class. `Strch` returns an index into the string given as the first argument of the character given as the second argument, or `NULL` if the character is not in the string. Thus it can be used to classify characters. The problem with this approach is that it is slow. It can involve the function calling overhead and even in compilers that implement it with in-line code the average time needed for testing if a character belongs in an n character class is $O(n/2)$.

The problem was solved by developing a small tool; the *character type compiler*: `ctc`. `Ctc` takes as input a file containing sets of recognition macros the user wants to define and regular expressions defining the characters for which the macro should return true. The file is line based. Comments starting with # and blank lines are allowed. The following lines represent a portion of the input file used for Haskell tokens.

```
# Rest of an identifier
R isidrest [A-Za-z0-9_']

# SYmbol start
Y issymstart [!#$%&*+./<=>?@\\^|~]
```

Based on the input the compiler creates a C file containing a bit vector table and the suitable macros. The macros take a character as input and use the character to index into the bit vector table. An *and* operation is performed on the table element based on the character class. Different character classes are represented by different bits being set. In this way with typically two CPU instructions a character can be portably classified. The character class compiler was itself written in the *Perl* programming language [Wal88].

1.5.2 Character Copying

In a typical *lex* based lexical analyser a character is copied three times before arriving in dynamically allocated memory. First the character is copied from the kernel buffer memory into a buffer of the *stdio* library. From the *stdio* buffer it is then copied to the internal *lex* array `yytext`. When an identifier needs to be saved, dynamic memory is allocated and it is finally copied into that memory. The copying from the *stdio* buffer to the data area of *lex* is particularly expensive since it is implemented as two levels of macros, the `getc` macro of the *stdio* library and the `input` macro of *lex*.

The method used involved the implementation of an alternative input output library. The design of the *fio* (fast input output) library [Hum88] outlined in [Hum88] offered a very efficient alternative. Access to the *fio* buffer is directly done via a pointer, in a line by line fashion. Since most tokens in HASKELL can not span lines this design was found to be suitable. The input part of *fio* was implemented and separately tested. It was found to be significantly faster than *stdio*.

A special macro, `input`, for interfacing to the *fio* library was created. The tasks of this macro are:

- Taking characters out of the *fio* buffer.
- Counting lines (needed for error reporting).
- Maintaining a column pointer and expanding tabs (needed for the layout rules).

- Checking for end of file.
- Cooperating with the one character pushback facility.

Through very careful coding the operations above are typically performed in 14 CPU instructions. [1]

1.5.3 Memory Allocation

Tokens that form identifiers need to be saved from the scanner buffer into dynamically allocated memory so that they will not be overwritten by new input. Memory allocation, especially when allocating many small blocks, is an expensive operation. For this reason an alternative approach was used. A big hunk of memory was requested that was then treated like a stack. Identifiers are put into it and terminated by a trailing 0. A stack pointer points the the first free character in the stack, while a counter determines characters remaining. When the stack is filled a new one is allocated.

1.5.4 Symbol Table Updates

Every symbol encountered is placed directly into the symbol table. The way the symbol table is organised ensures that if the symbol is already defined, no additional entry will be made. The symbols are placed into the symbol table directly from the lexical analyser input buffer. Subsequently the symbol table entry is compared against the address of the symbol buffer. If the symbol is located in the symbol buffer it is then copied to dynamically allocated memory as described above. In this way unnecessary copying and memory allocation and freeing are avoided.

1.5.5 The Ultimate Combination

The three types of improvement outlined above present the possibility to combine the tasks of reading a token, recognising it, and saving it into a single task! Many tokens, like user defined identifiers can be recognised by repeated uses of one of the *ctype* macros described. When the scanner finds an identifier it transfers control to the `save` macro. The `save` macro copies characters from the *fio* buffer to the dynamic memory stack (adjusting the stack pointer) while a given character classification macro (which is passed as a parameter) holds. In a typical case

[1] Most common execution path of the `input` macro compiled for the Intel 80286 family using the Microsoft C compiler version 5.1 with maximum optimisations.

an identifier is recognised and copied to dynamic memory and all counters are maintained using 28 CPU instructions per character.

1.5.6 Performance

The resulting scanner when applied to the Haskell standard prelude scanned at a speed of 7800 tokens per second (128μs per token) [2]

With a typical instruction timing for this CPU configuration of 3μs this averages to 420 CPU instructions per token. This is consistent with the 28 instructions needed for scanning an identifier character. Identifiers are typically 6 characters in length and one has to allow for operating system overhead, *fio* library overhead, memory management and scanning of more complicated tokens. (Identifiers are the second fastest token to scan, the fastest being single characters. Care to their implementation was taken because they occur very frequently.)

The previous implementation using *lex* was not completed to scan all the token types and thus a comparison of the performance of the two is not very meaningful since the *lex* implementation would become slower if it were to implement all the functionality that was put into the C based scanner. A rough insight to the performance improvement can be obtained by the fact that the (deficient) *lex*–based scanner was working at a speed of 4500 tokens per second.

1.6 Testing

The lexical analyser was tested using a specially built test harness. The harness repeatedly calls the lexical analyser and displays the returned token and its associated value. Tokens are displayed as symbolic constants and not as numbers making the output much easier to read. This is achieved by reading the token definition file `y.tab.h` into a hash table when the program is run. This automatic run-time configuration of the test harness ensures that it is always up-to-date.

During the testing 9 errors were found in the *lex* generated lexical analyser before it was made obsolete by the C hand crafted version. Up to now 31 errors have been found in C version with their rate of incidence constantly diminishing. Using the reliability index formula given in [Per87, p. 2.5] this gives an RI of 97%.

[2] All timing and profiling tests were performed on a COMPAQ DESKPRO 386/20e running COMPAQ Personal Computer DOS 3.31. The programs were compiled using the Microsoft C compiler version 5.1.

Chapter 2

Parsing

2.1 Technical Overview

The subject of language grammars, parsing and syntax directed translation has received the attention of hundreds of monographes and articles. An introductory discussion of syntax analysis can be found in [ASU85, pp. 159–278] and of syntax directed translation in [ASU85, pp. 279–342].

I decided to use *yacc* [Joh75] for the implementation of the parser as it would provide an organised framework for basing the parser, generate efficient code and allow for error recovery. Error recovery is very important in HASKELL as it is needed for implementing the layout rule! A general overview of LR parsing, the algorithms used by *yacc* and error recovery can by found in [AJ74].

HASKELL is a very difficult language to parse. A more general critique of this aspect of the language is given in section C. Problems I encountered, have been addressed at the implementation of various—by modern standards—baroque languages. A way to handle operators with varying syntactic rules of the APL language is given in [Str77]. The use of a two-level grammar to handle the complexity of the language PL/I can be found in [Mar84]. This scheme, described in section 2.3.5, is used to handle a shift-reduce conflict in the HASKELL grammar associated with the parsing of list comprehension qualifiers.

HASKELL features infix operators with user defined precedence and associativity. Handling them in a *yacc*-based grammar is very difficult. A way for generalised parsing of distfix operators is given in [Jon86]. Jones (who is also a member of the HASKELL committee) used this scheme in a modified version of the Sasl functional language. Jones' scheme provides the basic idea of having a special token value associated with all operators with a special syntactic feature (infix operators are a

specialised case of distfix operators). He does not however address the problem of varying precedence and associativity. Thus the scheme was extended in a way described in section 2.3.1.

Expression juxtaposition without an operator as used in HASKELL is also found in the *awk* programming language [AKW88] for indicating string concatenation. That feature is responsible for 156 shift-reduce errors in the Eighth Edition [V885] *nawk* grammar. The authors have attempted to fix this by adding a `%prec` keyword in the *yacc* specification rule for string concatenation. This method does not work for reasons explained in section 2.3.1. Running *yacc* on the unaltered source and on the source without the precedence specifier produces the same transition tables. [1] The source for the project `GNU` reimplementation of *awk*, *gawk* [CRRS89] contains exactly the same fruitless attempt.

The C language is not LALR(1) because of the `typedef` keyword. An implementation note in [HJ87, p. 118] explains that C compilers based on *yacc* such as the portable C compiler [Joh82] deal with this problem by feeding information back to the lexical analyser. The same approach was used for some of the HASKELL grammar problems.

As discussed in section 2.4 the most efficient way to build a tree is to build it in reverse order and, when finished, reverse the whole structure. The list reversal algorithm used, and a very elegant abstraction for implementing it is given by [Suz82].

Finally some limited discussion of module implementation can be found in [Wir77].

2.2 General Description

The parser is contained in a single *yacc* source file, `parse.y`. The file contains the type declarations for all the possible nodes and tokens, the grammar rules and three node building functions. The grammar is very loosely based on the one that appears in [HWA+90, pp. 116–120]. A number of rules were changed in order to remove ambiguities. The ambiguities were removed by the following techniques:

- Restructuring of grammar rules.
- Addition of new tokens and closer interaction with the lexical analyser.
- *Yacc* precedence rules.

[1] The number of shift-reduce errors is increased from 154 to 160, but the default actions of *yacc* remain the same.

- Merging of grammar rules and addition of semantic analysis.

Most tree nodes are built on the fly, since they are not built in more than one place. A special C macro, `defstruc`, is defined that creates a tree node of the appropriate size and type on the heap.

2.3 Handling the Grammar Ambiguities

2.3.1 Definable Operators and Function Applications

The HASKELL grammar contains two features that are very difficult to parse. The user may define new infix operators and constructors using the `infixl`, `infixr` and `infix` keywords. When defining those keywords the user specifies their associativity (left associative, right associative or non associative) and their precedence level from 0 to 10. This associativity and precedence defines how expressions and patterns will be parsed at compile time. This is a major problem since the behaviour of the parser needs to be changed during parse time. For example with a declaration `infixl 3 -` the expression `a - b - c` must be parsed as `((a - b) - c)` whereas with a declaration `infixr 3 -` the same expression must be parsed as `(a - (b - c))`.

Fixed precedence and associativity of expression operators can be handled very elegantly in *yacc* by using the *precedence* rules (`%left`, `%right` and `%noassoc`). As an example a language for a simple calculator can be defined as:

```
expr : expr '+' expr
| expr '-' expr
| expr '*' expr
| expr '/' expr
| '-' expr
| '(' expr ')'
| tNUMBER
;
```

This grammar is ambiguous. A number token can either be reduced to an expression or the parser state can be shifted to a state waiting for an operator. It can be dissambiguified either by splitting the rule into many subrules (expr, term, factor) or by adding the following rules at the beginning of the *yacc* grammar:

```
%left '+' '-'
%left '*' '/'
%left uminus
```

The rule for unary minus needs to be changed to `| '-' expr %prec uminus`. These changes have the effect of associating a precedence and an associativity with each possible input token. In addition each rule inherits the precedence and associativity of the last token with defined precedence found in its body.

In the case of a parsing conflict *yacc* behaves as follows:

- If no precedence is associated with the rule than an error is reported when creating the parse tables.
- If there is a precedence associated with the two rules that create the conflict then the action dictated by the rule with the highest precedence is performed.
- If both rules have the same precedence then the action depends on the associativity of the input token:
  - If the input token is left associative then a reduce is performed.
  - If the input token is left associative then a shift is performed.
  - If the input token has no associativity related to it then a parse error is generated.

The user specification of the associativity and fixity of operators and constructors can be handled by defining a series of symbolic operators of fixed precedence and associativity for all possible specifications. These are defined as tokens. When the user specifies the fixity of an operator or constructor, the parser updates the symbol table specifying giving a new token value to the token. Every time the lexical analyser encounters the token it checks the symbol table to see if its value has been redefined. If it has it returns its new value, else it returns its default value.

This approach would mean the addition of 30 tokens for the operators (10 precedence levels and 3 kinds of associativity) and 30 for the constructors. Doing this produced some obscure bugs. It turned out that *yacc* has a fixed size table for recording precedence rules which overflowed without a warning into another data structure. By merging the specifications for the constructors with those of the operators the problem was eliminated.

The existence of currying in HASKELL creates an additional problem when parsing. The HASKELL syntax defines function application as the textual juxtaposition of a function to its arguments. As functions are first class citizens in HASKELL this is in effect the juxtaposition of two expressions. This operation has a specific precedence level [HWA+90, p. 10] and binds left to right.

Since no token is associated with function application *per-se* the precedence and binding can not be specified directly. Using the `%prec` keyword in association with the rule (as misguidedly done in the *awk* source).

This precedence can not be enforced using the `%prec` keyword. As explained above the disambiguifying values are associated with specific tokens. The presence of a `%prec` keyword alters the precedence of those tokens. Since an expression can start with a token other than an operator the precedence rule will not come into effect for those other tokens. The solution to this problem is to find the $FIRST$ set of the expression and give the requisite precedence to its members. The set $FIRST(\alpha)$ for a string of grammar symbols α is defined [ASU85, pp. 188–189] to be the set of terminals that begin the strings derived from α. The following rules can be used for determining $FIRST(X)$ for a grammar symbol X:

1. If X is a terminal, then $FIRST(X)$ is $\{X\}$.
2. If $X \rightarrow \epsilon$ is a production, then ϵ is added to $FIRST(X)$.
3. If X is a nonterminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$ is a production, then for all $\alpha_k = FIRST(Y_k)$, α_k is placed in $FIRST(X)$ if $\forall i < k Y_i \stackrel{*}{\Rightarrow} \epsilon$

Applying the rules above to `expr` yields the set:

```
tVARID tCONID tCONIDCLS tCONIDCON tINT tDOUBLE tCHAR tSTRING '(' '['
```

All the members of the set were given the appropriate precedence and binding for function application. In addition a sentinel `Pfuncap` was also given the same precedence to use in conjunction with `%prec` in the rule for unary minus.

2.3.2 General

In this section I explain the major techniques used for handling grammar ambiguities. A step by step account on how all the problems were eliminated is given in sections 2.3.3 and 2.3.5.

2.3.3 September Version

The grammar as given in the September 89 document generates 22 reduce reduce errors and 106 shift reduce errors. The changes made to the grammar and their effects on the number of errors are given in table 2.1. Further explanations are given bellow.

1 The lexical analyser was made to return different tokens if an identifier identifies a class or another object.

2 The specifications for `varfun` and `infun` were:

Change	Errors	
	shift / reduce	reduce / reduce
Initial	106	22
Lexical tie [1]	102	22
Import list restructuring	101	22
`varfun` and `infun` [2]	99	22
Lexical tie [3]	99	16
Recursive `atypelist` [4]	83	16
Conflicting `atypelist` [5]	75	9
Precedence rules for `exp` and `aexp` [6]	23	9

Table 2.1: Changes to the September Grammar

```
infun <- infun ';' infun | ...
```

The intent was probably:

```
infundef <- infun | infundef ';' infun
infun <- ...
```

3 The lexical analyser was made to return a different token according to the arity of constant identifier type `tycon`.

4 The grammar definition for `atypelist` was not left recursive. It was converted to left recursive form as recommended in [Joh75].

5 The grammar included two conflicting definitions for `atypelist`. They were split into `atypelist0` and `atypelst1` for cases where different arity is expected.

6 The two grammar rules for expressions `exp` and `aexp` were merged. *Yacc* precedence (`%left`, `%right` and `%noassoc`) disambiguifying rules for all expressions. The rule specifying function application was extremely difficult to dissambiguify. The disambiguifying mechanism of *yacc* specifies that the token or literal appearing in the rule must have a precedence and binding type associated with it. The function application grammar rule has no literal or token. Specifying a precedence using the `%prec` keyword is not possible since the precedence given with `%prec` is used to override the precedence of any other tokens or literals. The solution adopted was to give the appropriate precedence to all literals that could form the beginning of a new expression.

Two shift of the remaining reduce conflicts are part of the language. The definition of `topdecl` allows a context to precede the class and the instance definitions. However a context can contain class names, which also occur in the class and instance definition expected.

At this state, the April version [HWA+90] of the HASKELL report was published.

2.3.4 April Grammar changes

The changes made in the grammar by the April report were so numerous that the grammar had to be written from scratch. The changes that affected the grammar are:

- Many of the names used in the syntax description changed. I had tried to stay close to the terminology used, and found that most terms used had changed. (e.g. `imports` was changed to `impdecls`).
- Triple dot which indicated that a whole module was exported changed to double dot.
- The precedences of conditional expressions and where expressions were changed.
- The things that could be exported changed from a list of names to a more specific and limited list. The ability to export type constructors and type classes as a whole unit was introduced, while the syntax for exporting a whole module changed so as not to require brackets before the module name.
- Syntax for renaming changed. The token `to` was changed to the token `as`.
- Type declarations were apparently redesigned from scratch:
  - Tuple declarations went away.
  - The class derivation was introduced.
  - The expose type declaration disappeared .
  - The where clause in instance declarations became optional.
  - The default declaration was introduced.
  - In low level type declarations the variable and identifier definitions were removed.
  - The unit type was introduced.

- Brackets were introduced around the class list in a context.
- A vertical bar symbol was added to separate constructors in constructor lists
- The class declaration using a double colon was removed
- A new syntax for class instances was introduced.
- Else is no longer optional in expressions
- Case and expression type signature was was moved from `aexp` to `exp`.
- The empty expression () was introduced
- Qualifiers were made optional
- The successor pattern and the unit patterns were introduced.
- The new tokens `to, hiding, default` and `deriving` were introduced.
- Fixes were made compulsory.

2.3.5 April Version

The grammar as given in the April 90 document generates 24 reduce reduce errors and 113 shift reduce errors (more than those in the September document). An outline of the progress in removing the problems from the grammar is given in table 2.2. Details are given in the list bellow (the numbers in brackets indicate the number of shift-reduce and reduce-reduce errors after each change).

Sat May 19 11:01:45 (112,24) In the rule for `topdecl` moved the rule for `optcontext` directly into the instance declaration.

Sat May 19 11:05:22 (112,17) In the rule for `pat` made the `-` in from of integer mandatory. The case where the minus is not mandatory is covered by `apat` which can be a `literal` which can be an `integer`. This removed seven reduce-reduce conflicts.

Sat May 19 11:08:50 (111,17) In rule for `decl` was modified by merging the `optcontext` rule.

Sat May 19 11:13:09 (109,15) In rule for `topdecl` the type list in brackets alternative was in conflict with the `atype` unit type, parenthesized type and tuple declarations.

Change Date	Errors	
	shift / reduce	reduce / reduce
Initial	113	24
Sat May 19 11:01:45	112	24
Sat May 19 11:05:22	112	17
Sat May 19 11:08:50	111	17
Sat May 19 11:13:09	109	15
Sat May 19 11:26:25	108	15
Sat May 19 11:29:48	107	15
Sat May 19 11:36:18	107	15
Sat May 19 11:42:30	106	15
Sat May 19 11:52:30	74	15
Sat May 19 12:04:24	140	15
Sat May 19 12:14:48	160	21
Sat May 19 14:28:58	352	21
Sat May 19 14:33:57	37	30
Sat May 19 14:39:21	2	30
Sat May 19 14:52:18	1	0
Sat May 19 14:58:27	0	30
Sat May 19 15:13:04	0	0
Sun Jun 09 22:51:40	0	3
Sun Jun 10 13:33:03	1	0

Table 2.2: Changes to the April Grammar

Sat May 19 11:26:25 (108,15) Gave precedence to `tCONOP` to defined how the sequence:

```
pat1 conop pat2 conop pat3
```

should be parsed using the `pat` rule.

Sat May 19 11:29:48 (107,15) Moved the `optcontext` rule explicitly into the `exp` rule.

Sat May 19 11:36:18 (107,15) For the rule `aexp` moved an extra rule that defined the optional second expression in an arithmetic sequence into the rule. This change did not change the number of errors.

Sat May 19 11:42:30 (106,15) Restructured the `aexp` rule. Made the lists of 0, 1 and 2 elements explicit so that they would not conflict with the start of an arithmetic sequence and then allowed for an `exprlist3` followed by more expressions.

Sat May 19 11:52:30 (74,15) In the rule for `exp` changed the expression type signature derivation from

```
exp :: [context =>] type

aexp :: [context =>] type.
```

The former case was ambiguous: is `exp + exp :: X` interpreted as `(exp + exp) :: X` or as `exp + (exp :: X)`. It turns out that section 3.11 of the manual specifies `aexp` instead of `exp`.

Sat May 19 12:04:24 (140,15) Replaced all occurrences of `con` with `conid` and all of `var` with `varid`. The result was disappointing. Although the two were supposed to be the same I had defined `con` wrongly as only `tCONCLS` and thus a number of reduce reduce errors were hidden.

Sat May 19 12:14:48 (160,21) Replaced all occurrences of `conid` with `tCONID | tCONIDCON | tCONIDCLS`.

Sat May 19 14:28:58 (352,21) Removed the function application rule to see the effect. This change clearly demonstrates the ambiguity that arises from curried function application.

Sat May 19 14:33:57 (37,30) Changed all occurrences of `varid` to `tVARID`. Changed `literal` in the definition of `apat` to its constituents `tINT tSTRING` etc. In this way a "precedes" set for `aexp` which contains all the tokens that can start an `aexp` was formed. This set was placed in a precedence disambiguifying rule for function application.

Sat May 19 14:39:21 (2,30) Corrected the "precedes" set used for function application. Had used ']' instead of '[' as the start token character of lists.

Sat May 19 14:52:18 (1,0) Removed the list comprehension rule from `qual` to confirm its effect on the reduce-reduce errors. The hypothesis was that all of these errors were a result of that rule. A solution would need a very complicate and unstructured lexical tie, so if this rule was not a major contributor another solution could be found. The hypothesis was verified, so the lexical tie was introduced.

Sat May 19 14:58:27 (0,30) Introduced a `quallist` in the rule for `qual` to remove the ambiguity over how `,` would associate.

Sat May 19 15:13:04 (0,0) Added back the `qual` rule and the lexical tie. A new pseudo-token was introduced `tPATCONTEXT`. This token does not represent a real lexical item. It represents a context where the lexical analyser has done a bit of scanning ahead as instructed by the parser by the `check_patcontext()` function to see if there is a `tLARROW` on the current scope level in the next tokens scanned up to a `']'` or `','`. This is quite complicated since the lexical analyser must invoke itself to remove comments, work out the layout rule etc. Furthermore all tokens scanned must be pushed back in some sort of stack with a `tPATCONTEXT` possibly on the top of the stack. For all this the lexical analyser must be reentrant. Since the lexical analyser is called by the parser and not the opposite it must also have state associated with it. This structuring conflict was resolved by adding one more level of function indirection to the lexical analyser.

Sun Jun 09 22:51:40 (0,3) Added three errors in order to parse correctly successor patterns. The rule specifying the successor pattern was never used. When a variable identifier was found it was directly reduced to an `apat`. This happened without a warning because of the precedence rules given to `TVARID` in order to resolve function application ambiguities in expressions. The solution tried was to replace the `tVARID` with a `apat`.

Sun Jun 10 13:33:03 (1,0) By specifying a more general rule for successor patterns (their left and right hand sides can be patterns) and adding semantics checking reduced the conflicts to a single shift/reduce. This occurs in the case where a pattern appears on the left hand side. In that case the left hand side can contain variable operators. Thus `x + 1` can either be parsed as two patterns separated by the + operator or as the successor pattern of x.

2.4 The Parse Tree

The result of applying the HASKELL grammar on a set of files is a parse tree. The parse tree is a tree of structures representing different syntactic elements of HASKELL . In places where a single syntactic element could have several underlying representations (e.g. an expression can be an integer or a function) a union within the structure was used to allow the different types to be stored in the same place.

In some cases different objects of the same type have different memory requirements. In theory memory can be conserved by using a pointer to a dynamically allocated data structure of the appropriate length. However, the differences in memory requirements were small (typically no more than 8 bytes) and for some implementations of the dynamic memory allocator the allocator overhead (12 bytes in the one described in [KR88, p. 185]) was higher than the actual space savings.

More space savings were achieved by combining lists and single items. In many cases single items are the exception, and usually they are composed into lists. In those cases the overhead of separately creating and managing lists of items was reduced by creating the items as lists of one element for the beginning.

Enumerated types are used to distinguish differing objects. These offer increased functionality compared to `# define`'s in type checking and debugger use.

All lists in the tree have the next pointer as the first element in the structure. This allows for general functions on lists to be written. Lists are built in the reverse order. Without any special pointers (which take extra space and time to maintain) adding an element at the end of a list of length n takes n operations. Thus building a list of length n has an overhead of $n(n+1)/2$. On the contrary a list of n elements, once built, can be reversed in n operations.

The tree reversal is performed at the end of the parsing. An automatically generated program traverses all the tree. It distinguishes between linked list anchors and linked list intermediate pointers. When a list anchor is found the list reversal algorithm is applied to it. Then the tree reversal operation continues for each element of the list. The program is automatically generated by a *perl* script based

on the tree definition header `tree.h` in a manner similar to the one described in section 2.10.

In some cases in the grammar, syntactic sugar is translated into concrete HASKELL on the fly. For example `- exp` is translated to a function application of the `negate` function.

2.5 The Mini-Parser

As indicated in section 2.3.5 in some cases the *lexical analyser* needs to parse ahead to determine in what context the parser is in. In order to do this a mini parser was developed. This parser, written in C, uses precedence rules to determine how far to parse. Round, square and curly brackets are counted and matched. Since its input comes from the lexical analyser white space and comments are automatically removed. The *follows* set [ASU85, p. 188] for the specific sentential form is used to terminate the parse ahead.

Since the *mini parser* does not do any real work the tokens it consumes must be placed back for delivery to the real parser. In addition if it finds the context to be a pattern context then the pseudo token `tPATCONTEXT` must be placed in the beginning of the token list. Furthermore it is possible for the mini parser to be called before the list of tokens that have been stored from a previous invocation has been exhausted. For these reasons a data structure is needed that can have items inserted on both ends, and removed from one end. Additions and removals from the data structure should be able to be intermixed. In order to achieve the desired effect a ring buffer was used. Two pointers indicate the beginning and the end of the data. The buffer is empty when the two pointers coincide. (Naturally a test is made to check if the buffer overflows each time an item is added). The buffer contains the token number of each token that is consumed and additionally its *value*. This is set be the lexical analyser into the `yylval` union and is the copied in the buffer.

Calls to the scanner are passed through an additional layer of code which checks the ring buffer. If there is an item in the ring buffer then that item is removed and returned. In all other cases the value of the real lexical analyser is returned.

2.6 The layout rule

As explained in section 1.3 not all layout rules can be resolved by the lexical analyser. The rule that specifies that a brace needs to be inserted whenever the next item is illegal, but a brace would be legal is more easily handled by the parser. (It

can be handled by the scanner by forming the *precedes* set for the closing brace, the *follows* set for every item for the *precedes* set and returning a closing brace between every pair of tokens (α, β) where $\alpha \in precedes(\{) \wedge \beta \notin follows(\alpha))$.

The solution adopted was based on *yacc*'s error recovery mechanism [Joh75, p. 25]. The token } was substituted by the rule `close_brace`. The `close_brace` rule can either be literal closing brace or a *yacc* error production. A *yacc* error production is exactly the token that would be legal in the case where the next token is illegal. This fits precisely with the description of the layout rule. When the error production is invoked the lexical analyser is informed by calling the `close_brace` procedure. This is done in order to remove the pending closing brace form the lexical analysers stack. After the error production the *yacc* macro `yyerrok` is called to allow the trapping of any other errors. (*Yacc* will otherwise suppress any errors unless four tokens have been successfully parsed. This is not acceptable in this case.)

2.7 Dealing With Interface and Implementation Modules

The interface modules are mostly a subset of the implementation modules. Duplicating the grammar rules for the interface modules seemed unnecessary duplication of effort and code, so I decided to use the implementation module rules and do some semantic checking within the rules. For this reason a global variable was added `module_type` which contained the type of module that was parsed. This variable is set *within* the grammar rules (i.e. in code inserted between the yacc tokens), as there is the only place in the grammar where one can distinguish between the two types of modules. In order to allow for the abbreviated module the open brace of the body was moved in the `module` production rule.

2.8 Lexical Ties

Up to now three lexical ties have been described:

1. The addition of a pseudo-token returned by the lexical analyser by the request of the parser to find if a qualifier is a pattern or an expression (section 2.3.5).

2. Informing the lexical analyser that a brace was added by a grammar rule so that the scanner could adjust the layout tables (section 2.6).

3. Changing the token value of the operators and constructors after the user defines their associativity and precedence (section 2.3.1).

Token	Where
aconid	modid
con	constr
conid [1]	export
conop	op
literal	aexp
modid [1]	export
tycls [1]	export
tycon	export
tyvar [1]	atype
var	aexp
varid	export
varop	op

Table 2.3: Tokens Appearing in the HASKELL Grammar

Token	Synonym	Representation (example)
var	varid	name or (+-)
con	conid	Name or (:+-)
varop		+- or ‘name‘
conop		:+- or ‘Name‘
tyvar	avarid	name
tycon	aconid	Name
tycls	aconid	Name
modid		Name

Table 2.4: Token Synonyms

I addition to these a mechanism is needed to classify the tokens to different types. Table 2.3 contains a list of the tokens used in the grammar.

The number of tokens used in the grammar is a source of confusion as many are synonyms. Table 2.4 contains the token synonyms as defined in the lexical grammar, and some examples.

From table 2.4 the following minimal set of tokens needs to be distinguished:

1. con Constructor.
2. var Variable.
3. conop Constant operator.

4. varop Variable operator.

5. tycon Type constructor.

6. tyvar Type variable.

7. tycls Type class.

8. modid Module identifier.

From the above list and the fact that 1, 5, 7, 8 and 2, 6 belong to different name spaces [HWA+90, section 1.4] we can conclude that the lexical analyser can not — in general— detect in which of (1, 5, 7, 8) or (2, 6) an identifier belongs through semantic analysis (e.g. by looking at the symbol table) as there might be four different meanings for it. The only case where a semantic analysis will help is in distinguishing between a type class and a type constructor which are limited in the same namespace.

From the above we conclude that at any time in the grammar there should be only one of:

```
con
tycon tycls
modid
```

or one of:

```
var
tyvar
```

A semantic check needs to be done for `tycon`, `tycls` and `modid` to verify that they do not start with a bracket (as `tCONID` is allowed). The same is true for `tyvar` with respect to `tVARID`.

In order to distinguish between `tycon` and `tycls` and all the others the following lexical tie-in is used:

tCONIDCON Returned when the name is a type constructor in that scope.

tCONIDCLS Returned when the name is a type class in that scope.

tCONID Returned in all other cases.

The type of `conid` returned by the lexical analyser does not force the token to be interpreted like one of `tycon` or `tycls`, but serves as an indicator in cases where the distinction is relevant. For example the rule for `modid` is:

```
conid : tCONIDCON | tCONIDCLS | tCONID ;
modid : modid ;
```

As `modid` is in a different name space than `tycls` an identifier that represents a type class shall and will still be interpreted as a module name in the context where a module name is needed. On the contrary the rules for `tycls` and `tycon` are:

```
tycls : tCONIDCLS;
tycon : tCONIDCON;
```

In addition to this the grammar needs to be modified so that when a new class or constructor is defined `conid` is used instead of `tycls`.

2.9 Symbol Table

The symbol table is the place where all non reserved identifiers are stored. It is organised as a hash table of binary trees. From data presented in [LV73], it appears that a hash table is a viable technique for organising a symbol table if another mechanism is available for resolving hash collisions.

The first character of the symbol is used as the index into the hash table. A special opaque data type the `stab_entry` has been defined as the handle in conjunction with symbols. In addition the data type `stab_table` provides the opaque data type definition of the symbol table. Access procedures for adding new symbols, accessing existing symbols, walking through the table using a higher order function, adding "floating" pseudo-symbols (used by the type checker) and getting the symbol values are defined.

2.10 Testing

In order to test the parser a way to view the parse tree was needed. Since during the initial phase of the implementation the tree was a moving target it was decided to automatically create the program to print the tree. This was not very difficult as care had been taken during the coding of the tree to write it in such a way that a mechanical translator based on regular expressions could parse it. Thus the layout rules were followed scholastically, enumerations were declared immediately before the structure in which they were used, and complicated syntax was avoided.

A 239 line program written in the Perl programming language [Wal88], was developed to create a tree printing program out of the tree description. The tree

printing program recursively walks through the tree displaying the values of the enumerations and all basic types. Indentation is used to pictorially represent the tree structure. The value of the enumerations is used to determine which member of the union to print. From a 470 line description of the tree (`tree.h`) a 1062 line file of C code is generated. Clearly the effort put into building this tree printing tool was worth it. In addition as less programs had to be manually modified each time the tree was changed, changes for reasons of efficiency or clarity were easier to make.

A specially built test harness was built that would call the parser and display the syntax tree. During the testing 11 errors were found in the parser and its associated modules. Using the reliability index formula given in [Per87, p. 2.5] this gives an RI of 99.6%. The index is substantially better than the one achieved for the scanner. The reasons for this are probably:

- *Yacc* found many errors at compile time.
- No tricky optimisations were tried (other than the list reversing).
- Once the conflicts are removed from the grammar the technique for creating the tree is straightforward and well understood.

Chapter 3

Modules and Prelude

3.1 Technical Overview

A commonly used language based on modules is *Modula-2* [Wir85]. A description of its implementation can be found in [Wir77]. The HASKELL report does not specify how the implementations and the interfaces relate to files and to each other. It leaves that area of specification to the *compilation system* [HWA+90, p. 37]. For technical reasons the current design requires every *implementation* module to have an associated *interface* module. Although not required by the system it is I do not think it is a good practice to have more than one module in a file. In this aspect the system, resembles most *Modula-2* implementations.

The standard prelude resembles the specification for the ANSI C hosted implementation run-time library [KR88]. As commented in [HJ87, p. 276] the C library headers and functions may be "built in" to the implementation and only exist in a virtual sense; library calls can be substituted with inline code. Naive implementations can of course define the library by using real header files and real functions. *Mutatis mutandis* the HASKELL prelude provides a functional specification for the run-time system. I decided to follow the naive way and use the actual prelude files, instead of building them into the language. The advantages of this approach are:

1. Ease of implementation. The language automatically acquires a suite of useful functions and types, once a minimal set of primitives has been implemented.

2. Modularity. The specifications for the utility functions exist outside the language system. They can be used by both the compiler and the interpreter.

3. Maintainability. A new version of the HASKELL report is due to appear on April 1991. Casting the prelude into efficient code at this stage is premature.

The major disadvantage of this approach is loss of efficiency. The system takes some time to read and parse the prelude every time it is started, and the functions could probably be implemented more efficiently as "built in" primitives. However the other two HASKELL implementations that I am aware of (Glasgow and Yale) are being written entirely in HASKELL [Hud89, p. 405], so this should not make a very big difference.

An attractive solution offered to the problem of lengthy initialisation is the *undump* method. This method is useful for programs that have to bootstrap themselves before being ready for user interaction. A virgin copy of the program is read on the store and starts executing the intialisation code. This typically involves reading parsing some external files. Once this is completed a complete core image of the file including its stack, heap and registers is dumped on the disk. Another program then converts that core image into an executable program in the state of the other program was before the dump. For example the sc EMACS editor [Sta84] has a user interface based on hundreds of lines of *Lisp* code. When the program is installed it reads all the lisp code and then performs the undump. In an analogous way the HASKELL front end can read parse and possibly type check the prelude and dump that state into a new initialised HASKELL executable.

3.2 Lexical Analysis

The first interface to the module system and prelude is presented at the lexical analysis phase. The lexical analyser merges the prelude and the user specified files into a single file that is passed as tokens to the parser. The usual interface of the `yywrap()` function is used for this purpose. When the end of a file is reached, the scanner calls the `yywrap` function to check if there is any more input. The `yywrap` function readjusts the current line number, column number and file name and allows the continuation of input.

3.3 Prelude Initialisation

The prelude is initialised by means of a secondary file that contains a specification of the files comprising the initialisation part of the language. This method decouples the prelude from the HASKELL system and allows the user to add or remove intialisation code. A user may elect not to read parts of the prelude (e.g. complex

numbers) or to substitute parts of the prelude with their own implementation. The file, called `prelude.i` is line oriented. Empty lines and lines starting with a hash character ('#') are ignored. The rest of the lines are assumed to contain a single file name that is loaded by the system before the normal processing of the user specified files.

At the time of this writing the system automatically includes a great part of the prelude on startup. The code included is a modified version of the code supplied with the HASKELL April report. The modifications are mainly the provision of an interface module for each implementation module (containing all the definitions of operators (with their fixity) types, classes and instances and all the type signatures), the removal of fixity declarations from the implementation modules (as they are given in the interface modules) and the correction of some minor syntax errors.

Chapter 4

Interpreter

4.1 Technical Overview

The factors one has to consider when choosing between an interpretation or compilation based implementation are given in [aNFV84]:

1. The larger distance in a compilation system between the source text and the generated code requires higher program complexity.
2. In a compilation based implementation the resulting execution will be more efficient as more checks are done by the compiler.
3. If the intermediate code used by the interpretation system is reversible then space savings can be made, by storing only the intermediate code.

The reasons listed above are dated (especially number 3). In addition some more reasons for choosing an interpreter are:

- Faster user interaction.
- Incremental system building, rapid prototyping.
- Provision of a language with meta-linguistic abilities (e.g. eval).
- Portability of implementation.
- Easy provision of code manipulation facilities such as source level debugging, profiling etc.

Interpreted language implementations tend to fall into three categories:

1. Command interpreters such as the UNIX shells *sh* [Bou86] and *csh* [Joy86]. An application of the functional programming paradigm in this area is *fsh* a functional command interpreter [Don87].

2. Interactive programming environments such as *Smalltalk* [Gol80], *InterLISP* [Tei78], the *I.C. Transformation Environment*, *QuickBASIC* [MSQ88] and *muProlog* [Nai84].

3. Little (and not so little) specialised languages such as the UNIX text processing family of *eqn* [KC], *tbl* [Les82], *troff* [Oss82] and *ditroff* [Ker], *awk* [AKW88], *perl* [Wal88] etc.

HASKELL is neither a command interpreter nor a specialised language, thus an interpretive implementation of HASKELL would fall into the second area, that of interactive programming environments. The need and importance of a user-friendly functional language environment is stressed in [Lei84] who outlines the experience of using the *InterLISP* environment. The various styles of *LISP* environments are summarised in [San78].

A formal classification of the various types of writing language for an interpreter is given in [LPT82, p. 802]. The ways interpretation can be implemented are categorised in [Kli81] as follows:

CLASS (Classical). The instruction at the address of the program counter is executed.

DTC (Direct Threaded Code). The instruction pointed by the address of the program counter is executed.

ITC (Indirect Threaded Code). The instruction pointed by the instruction pointed by the program counter is executed.

The *eval / apply* interpreter implemented can be characterised as a *DTC* interpreter since the execution pointer is moving on the tree nodes of the lambda tree.

An alternative approach to the benefits of interpretation is *throw-away-compilation*. A simple compiler generates code, usually on the fly, which is executed *in-situ*. The code is usually never saved. A comparison of these two approaches is made by [Rob83]. The throw-away compilation approach was particularly attractive for this project since a compiler was also written. The major implementation problem is that of portability. The UNIX operating system does not provide a portable way for the user to directly compile code into memory. A discussion of interpretation based on an intermediate language can be found in [KKM80].

The approach elected was a modification of the classical environment based *eval-apply* interpreter following the tradition started by [McC60]. The implementation of such an interpreter in *Hope* is described in [FH88, pp. 193 – 213] and in *Lisp* in [Lic86]. A more general implementation for abstract equations can be found in[HOS85]. A meta circular version of the interpreter (with one additional procedure `EVLIS` which evaluates a list calling `EVAL`) is given in [JSXX, p. 631]. The interpreter is based on the *SCHEME* language and is thus lexically scoped, exactly as required for HASKELL . That interpreter is then transformed into a statement oriented language presented in [JSXX, p. 637] suitable for VLSI implementation. Another description of a *Lisp* interpreter is given by [Bai85b]. I argue that my imperative implementation of the *eval-apply* interpreter resembles the SECD machine first presented in [Lan63].

In section 4.5 I describe a special compiler written to compile primitives from a HASKELL / C hybrid language into C / Examples of development and uses of such little languages are given in [Ben86]. The associative arrays used to store type information for the primitive compiler are featured in [Ben85]. The compiler was written in *Perl* [Wal88] a language containing all the features of *awk* (*awk* programs can be automatically translated into *Perl*. The suitability of *awk* for generating programs is demonstrated in [Wyk86]. Finally the concept of *yacc* like actions and the possibility of using a different language for them in a C environment is presented in [Set84].

4.2 General Description

The interpreter traverses sugared lambda calculus tree evaluating expressions. Special code hooks allow for debugging and profiling the performance of HASKELL programs. An extendable interfacing mechanism merges C and HASKELL types and objects allowing for easy addition of primitives and libraries. Finally a front end is provided for user interaction.

4.3 Interpreter Description

4.3.1 Lambda Tree

The parser operates on the lambda tree. The lambda tree which represents sugared lambda calculus is derived from the syntax tree by a series of transformations done by Tassos Hadjicocolis, namely pattern matching removal and lambda lifting.

A node of the lambda tree is a C structure with two main fields:

1. An enumeration type variable containing the type *tag* of the node.
2. A union containing the various types of nodes.

The tree initially found can contain the following types of nodes:

Function application. The node contains the function that is applied and the function it is applied to.

Variable. The node contains a symbol table handle pointer. Looking for that pointer in the correct environment or list of builtin functions will locate the value for the variable.

Recursive let. The node contains a pointer to a declaration list (variable = expression pairs) and the main expression.

Conditional. The node contains an expression to be evaluated, an expression to evaluate if the first expression evaluates to *true* and the expression to evaluate if the first expression evaluates to *false*.

Fatbar. The node contains two expressions α and β. The semantics of the result are:

$$\begin{array}{lcll} \alpha & \Box & \beta & = \alpha \\ FAIL & \Box & \beta & = \beta \\ \bot & \Box & \beta & = \bot \end{array}$$

Lambda abstraction. The node contains the lambda variable and the expression it is to be substituted in.

Error. This node occurs in cases of pattern match errors.

Tuple, Operand, Constructor The node contains the appropriate tag or symbol table entry.

Integer, Double, Character, String, Nil The node union contains the value of the item in a field of the appropriate type.

In addition to those nodes three more are added:

Suspension. The node contains a pointer to an expression and the environment over which it is suspended.

Closure. The node contains a lambda expression and the environment in which its variables are bound.

Operator. The node contains the arity, function and list of arguments to be passed to a builtin operator.

4.3.2 Environment

The environment is stored as a linked list of variable expression pairs. Since all the strings in the HASKELL system are guaranteed to be stored only ones, when the value of a variable is searched in the environment, only the pointers and not the actual names of the variables need be compared.

The circular environment needed for the evaluation of *letrec*s is constructed on the fly by creating a dummy environment node. Each (variable, expression) pair added to the environment as a suspension is made to be evaluated with that node as an environment. When the last pair has been added to the environment the dummy node is added as well. In this way a circle is formed in the environment.

4.3.3 Evaluate Code

Evaluation of an expression can be distinguished to the following cases:

Function application. The expression to be evaluated and the current environment are put into a suspension. The result of applying that suspension to the second expression is returned.

Variable If the variable is defined in the environment then the expression to which it is defined is evaluated and returned. If the variable is a builtin primitive then the arity and the function address for that primitive are fetched from the symbol table and placed in an newly created operator node. The argument list of that operator is set to empty.

Letrec. A circular environment is built as explained in section 4.3.2. The result of evaluating the expression in that new environment is returned.

Conditional. The first expression is evaluated. If the result is true then the result of evaluating the second expression is returned else the result of evaluating the third one.

Fatbar. If evaluating the first expression returns bottom then bottom is returned. [1] If evaluating the first expression does not return a fail node then the that result is returned, else the result of evaluating the second expression is returned.

Lambda. A closure node of the expression and the current environment is returned.

[1] This is to check if you were reading carefully. The sentence before the footnote is not to be taken seriously.

Suspension. The result of evaluating the suspension in the environment specified in the suspension is returned.

All other nodes. The expression is returned.

4.3.4 Apply Code

The code for evaluating function application is much simpler. There are only two cases:

1. The function to be applied is a closure. Then a new environment is created mapping the second expression to the lambda variable of the closure. The result of evaluating the closure lambda body in that environment is returned.

2. The function to be applied is an operator. The second expression is added to the list of arguments of the operator. If the arity of the operator is 1 then a C function defined in the operator is called with the list of arguments as an argument. Else A new operator note is created with arity one less than the current operator node.

4.4 Primitive library

In order to minimise the amount of coding I decided to implement the minimum number of primitives needed to provide all the features of HASKELL . Fortunately, this was also a goal of the HASKELL committee. (The reason the HASKELL committee provided a minimal number of primitives was in order provide a clean language definition). Thus the `PreludeBuiltin` part of the HASKELL *prelude* contains such a minimal set of primitives.

A brief description of the primitives implemented is contained in the following sections.

4.4.1 Type Conversion

`primCharToInt :: Char → Int`
Converts a character into a fixed precision integer. The result will always be positive.

`primIntToChar :: Int → Char`
Converts a fixed precision integer into a character. Integers with ordinal values that are the same as the characters of the machine character set are represented as those characters. Other integers are truncated in an implementation defined way. Usually the result corresponds to a binary *and* of a mask of ones, wide as the character representation and the integer.

`primIntToInteger :: Int → Integer`
Converts a fixed precision integer into a multiple precision integer.

`primIntegerToInt :: Integer → Int`
Convert a multiple precision integer into a fixed precision integer. If the value of the multiple precision integer is greater or less than the maximum or minimum fixed precision integer representable value the result is undefined.

4.4.2 Fixed Precision Integers

`primMinInt :: Int`
Returns the maximum fixed precision integer that can be represented on the system. The value is -2147483648 for 32 bit machines.

`primMaxInt :: Int`
Returns the minimum fixed precision integer that can be represented on the system. The value is 2147483647 for 32 bit machines.

`primEqInt :: Int → Int → Bool`
Compares two integers for equality. Returns true if they are equal, false if they are not equal.

`primLeInt :: Int → Int → Bool`
Returns true if the first integer is less than or equal to the second one, false if the second is greater than the first one.

`primPlusInt :: Int → Int → Int`
Returns the result of adding the two integers passed. If an overflow occurs the result is implementation dependent.

`primMulInt :: Int → Int → Int`
Returns the result of multiplying the two integers passed. If an overflow occurs the result is implementation dependent.

`primNegInt :: Int → Int`
Returns the negated value of the integer passed. If an overflow occurs the result is implementation dependent.

`primDivRemInt :: Int → Int → (Int,Int)`
Returns a tuple containing the quotient and the remainder resulting from dividing first integer passed by the second. A runtime error will occur if the second integer is zero.

4.4.3 Multiple Precision Integers

`primEqInt :: Integer → Integer → Bool`
Compares two multiple precision integers for equality. Returns true if they are equal, false if they are not equal.

`primLeInt :: Integer → Integer → Bool`
Returns true if the first integer is less than or equal to the second one, false if the second is greater than the first one.

`primPlusInt :: Integer → Integer → Integer`
Returns the result of adding the two integers passed.

`primMulInt :: Integer → Integer → Integer`
Returns the result of multiplying the two integers passed.

`primNegInt :: Integer → Integer`
Returns the negated value of the integer passed.

`primDivRemInt :: Integer → Integer → (Integer,Integer)`
Returns a tuple containing the quotient and the remainder resulting from dividing first integer passed by the second. A runtime error will occur if the second integer is zero.

4.4.4 Single Precision Floating Point

"The practical scientist is trying to solve tomorrow's problem with yesterday's computer; the computer scientist, we think, often

has it the other way round."
W. H. Press, B. P. Flannery, S. A. Teukolsky and W. T. Vetterling commenting on C 's half hearted support of single precision arithmetic in *Numerical Recipes in C* [PFTV88].

The primitives in this section are written in the C programming language using the standard C library. Traditionally C [KR78] did all single precision arithmetic by converting the values into double precision doing the operation and then converting the result back to single precision before storing. The reason for this is allegedly the difficulty of switching between the two modes on the PDP-11 computer. ANSI C [KR88] corrected this deficiency by specifying that single precision arithmetic is done in single precision. It does not however provide a single precision mathematical library. For this reason some operations in the following section are performed in single precision and some in double.

`primFloatRadix :: Integer`
Returns the exponent radix of the system single precision floating point representation. This value is 2 for IEEE arithmetic implementations.

`primFloatDigits :: Int`
Returns the maximum number of floating point digits that can be represented in the system single precision floating point representation. This value is 6 for IEEE arithmetic implementations.

`primFloatMinExp :: Int`
Returns the exponent of the smallest representable single precision floating point number. This number will be -37 for IEEE arithmetic implementations.

`primFloatMaxExp :: Int`
Returns the exponent of the highest representable single precision floating point number. This number will be 38 for IEEE arithmetic implementations.

`primDecodeFloat :: Float` $\rightarrow$ `(Integer,Int)`
Given a single precision floating point number x, returns a tuple (m, n) such that if b is the floating point radix $x = mb^n$. Also if d is the number returned by primFloatDigits m and n will be zero or $b^{d-1} <= m < b^d$ will hold.

`primEncodeFloat :: Integer` $\rightarrow$ `Int` $\rightarrow$ `Float`
Given a multiple precision integer m and a fixed precision integer n it primEncodeFloat returns mb^n where b is the single precision floating point radix.

`primEqFloat :: Float → Float → Bool`

Returns true if the two single precision floating point numbers passed are exactly equal.

`primLeFloat :: Float → Float → Bool`

Returns true if the first single precision floating point number passed is less or equal to the second one.

`primPlusFloat :: Float → Float → Float`

Returns the result of adding the two single precision floating point numbers passed. The operation is performed in single precision arithmetic. If an overflow occurs the result is implementation dependent.

`primMulFloat :: Float → Float → Float`

Returns the result of multiplying the two single precision floating point numbers passed. The operation is performed in single precision arithmetic. If an overflow occurs the result is implementation dependent.

`primDivFloat :: Float → Float → Float`

Returns the result of dividing the first single precision floating point number given, by the second one. The operation is performed in single precision arithmetic. If an overflow occurs the result is implementation dependent.

`primNegFloat :: Float → Float`

Returns the negated value of the single precision floating point number passed.

`primPiFloat :: Float`

Returns the single precision approximation to the geometric constant π.

`primExpFloat :: Float → Float`

Given a single precision floating point number x the result of the exponential function e^x is returned. The operation is performed in double precision arithmetic.

`primLogFloat :: Float → Float`

Returns the natural logarithm of the single precision floating point number passed. The operation is performed in double precision arithmetic.

`primSqrtFloat :: Float → Float`

Returns the square root of the single precision floating point number passed. The operation is performed in double precision arithmetic.

`primSinFloat :: Float → Float`
Returns the sine of the single precision floating point number passed. The operation is performed in double precision arithmetic.

`primCosFloat :: Float → Float`
Returns the cosine of the single precision floating point number passed. The operation is performed in double precision arithmetic.

`primTanFloat :: Float → Float`
Returns the tangent of the single precision floating point number passed. The operation is performed in double precision arithmetic.

`primAsinFloat :: Float → Float`
Returns the arc sine of the single precision floating point number passed. The operation is performed in double precision arithmetic.

`primAcosFloat :: Float → Float`
Returns the arc cosine of the single precision floating point number passed. The operation is performed in double precision arithmetic.

`primAtanFloat :: Float → Float`
Returns the arc tangent of the single precision floating point number passed. The operation is performed in double precision arithmetic.

`primSinhFloat :: Float → Float`
Returns the hyperbolic sine of the single precision floating point number passed. The operation is performed in double precision arithmetic.

`primCoshFloat :: Float → Float`
Returns the hyperbolic cosine of the single precision floating point number passed. The operation is performed in double precision arithmetic.

`primTanhFloat :: Float → Float`
Returns the hyperbolic tangent of the single precision floating point number passed. The operation is performed in double precision arithmetic.

`primAsinhFloat :: Float → Float`
Returns the inverse hyperbolic sine of the single precision floating point number passed. The operation is performed in double precision arithmetic.

`primAcoshFloat :: Float → Float`
Returns the inverse hyperbolic cosine of the single precision floating point number passed. The operation is performed in double precision arithmetic.

`primAtanhFloat :: Float → Float`

Returns the inverse hyperbolic tangent of the single precision floating point number passed. The operation is performed in double precision arithmetic.

4.4.5 Double Precision Floating Point

All primitives in this section are performed in double precision arithmetic.

`primDoubleRadix :: Integer`

Returns the exponent radix of the system double precision floating point representation. This value is 2 for IEEE arithmetic implementations.

`primDoubleDigits :: Int`

Returns the maximum number of floating point digits that can be represented in the system double precision floating point representation. This value is 15 for IEEE arithmetic implementations.

`primDoubleMinExp :: Int`

Returns the exponent of the smallest representable double precision floating point number. This number will be -307 for IEEE arithmetic implementations.

`primDoubleMaxExp :: Int`

Returns the exponent of the highest representable double precision floating point number. This number will be 308 for IEEE arithmetic implementations.

`primDecodeDouble :: Double → (Integer,Int)`

Given a double precision floating point number x, returns a tuple (m, n) such that if b is the floating point radix $x = mb^n$. Also if d is the number returned by primDoubleDigits m and n will be zero or $b^{d-1} <= m < b^d$ will hold.

`primEncodeDouble :: Integer → Int → Double`

Given a multiple precision integer m and a fixed precision integer n it primEncodeDouble returns mb^n where b is the double precision floating point radix.

`primEqDouble :: Double → Double → Bool`

Returns true if the two double precision floating point numbers passed are exactly equal.

`primLeDouble :: Double → Double → Bool`

Returns true if the first double precision floating point number passed is less or equal to the second one.

`primPlusDouble :: Double → Double → Double`
: Returns the result of adding the two double precision floating point numbers passed. If an overflow occurs the result is implementation dependent.

`primMulDouble :: Double → Double → Double`
: Returns the result of multiplying the two double precision floating point numbers passed. If an overflow occurs the result is implementation dependent.

`primDivDouble :: Double → Double → Double`
: Returns the result of dividing the first double precision floating point number given, by the second one. If an overflow occurs the result is implementation dependent.

`primNegDouble :: Double → Double`
: Returns the negated value of the double precision floating point number passed.

`primPiDouble :: Double`
: Returns the double precision approximation to the geometric constant π.

`primExpDouble :: Double → Double`
: Given a double precision floating point number x the result of the exponential function e^x is returned.

`primLogDouble :: Double → Double`
: Returns the natural logarithm of the double precision floating point number passed.

`primSqrtDouble :: Double → Double`
: Returns the square root of the double precision floating point number passed.

`primSinDouble :: Double → Double`
: Returns the sine of the double precision floating point number passed.

`primCosDouble :: Double → Double`
: Returns the cosine of the double precision floating point number passed.

`primTanDouble :: Double → Double`
: Returns the tangent of the double precision floating point number passed.

`primAsinDouble :: Double → Double`
: Returns the arc sine of the double precision floating point number passed.

`primAcosDouble :: Double → Double`
Returns the arc cosine of the double precision floating point number passed.

`primAtanDouble :: Double → Double`
Returns the arc tangent of the double precision floating point number passed.

`primSinhDouble :: Double → Double`
Returns the hyperbolic sine of the double precision floating point number passed.

`primCoshDouble :: Double → Double`
Returns the hyperbolic cosine of the double precision floating point number passed.

`primTanhDouble :: Double → Double`
Returns the hyperbolic tangent of the double precision floating point number passed.

`primAsinhDouble :: Double → Double`
Returns the inverse hyperbolic sine of the double precision floating point number passed.

`primAcoshDouble :: Double → Double`
Returns the inverse hyperbolic cosine of the double precision floating point number passed.

`primAtanhDouble :: Double → Double`
Returns the inverse hyperbolic tangent of the double precision floating point number passed.

4.5 Primitive Description Compiler

All primitives have some common characteristics. Each primitive needs to:

- isolate its arguments from the linked list passed,
- evaluate the arguments into weak head normal form,
- assert that the arguments passed are of the correct type (this is done in the debug version of the program to flag potential programmer errors),
- create a new node of the appropriate type,
- fill the appropriate field of the node with the result and
- return that node.

In order to avoid this repetitiveness I implemented a special language for describing primitives. This small language providing the interface between C and HASKELL contains features of both languages. It also uses some conventions such as the `$` pseudo-variable found in *yacc* [Joh75]. The primitive description file can contain comments starting with the # character and blank lines. Code between {% and %} pairs is literally included in the resulting C output. Its purpose is to allow for the specification of include files, global variables, data structures and procedures.

The user needs to specify a map between the HASKELL types, their C representations, the C enumeration constants used to represent them in an expression and the union field they belong to. Each item of the map starts on a new line with a `%type` keyword. Map elements are separated by s double colon. For example the map for fixed precision integer values is the following:

```
%type  Int      : int        : ex_int       : i
```

This means that a HASKELL value of type `Int` is stored in the structure union field `u.i` with the structure `kind` field set to `ex_int`. A C variable of type `int` can store such a value.

After the map is given the user can define the primitives. Primitives start on a line starting with the `primitive` keyword, followed by the name of the HASKELL primitive and its HASKELL type signature. A C block follows the primitive declaration. Within the block the pseudo-variables `$1`, `$2` etc. are substituted by the appropriate union fields of the real arguments, while the pseudo-variable `$$` stands for the correctly initialised result node. The definition for the `primEqInt` primitive might look as follows:

```
primitive primEqInt :: Int -> Int -> Bool
{
       $$ = ($1 == $2);
}
```

This is about 8 lines shorter and a lot clearer than the corresponding C code.

This meta-compiler is implemented in the PERL programming language. The associative arrays, variable substitution within strings and extended regular expressions of PERL significantly eased the implementation task.

Chapter 5

Code Generation

The purpose of this chapter is to describe the process that yields executable machine code, starting from abstract G code. The G code is produced by a series of transformations by Tassos Hadjicocolis (pattern matching removal, lambda lifting, supercombinators, G).

5.1 Technical Overview

The main reference for code generation from the G machine is [Jon87, 293–366]. Other abstract machines fo evaluating declarative languages are discussed in [JJ89] and [War83]. The Amber machine [Car85] is an example of a machine for direct implementation of functional languages. SK combinators and details on removing variable references are presented in [Tur79]. The need for highly specific abstract machine instructions to aid the code generation process is presented in [FH82]. The paper also contains ways for exploiting various machine specific pointer operations in abstract machines. Some of the advice given can be found in the 68000 code generator. A discussion on code generation for *Lisp*, assembler macros and various optimisations can be found in [GH81]. Assembler specific optimisations are given in [Han83].

Systematic optimisation of assembly code is done by means of peephole optimisers [ASU85, p. 554]. Some simple optimisations relevant to the code produced by the G translation process are given in [Jon87, p. 328]. Devidson [Dav84] generalises the technique of a simulated stack into a simulated cache. A more general discussion on constructing a peephole optimiser is in [Lam81].

The idea for a generic machine description file (and the file name suffix `.md` was take from the project GNU C compiler [Sta89]. The C calling sequence for

interfacing C with assembly language is presented and discussed in [JR81].

5.2 Machine Description Meta-generator

The problem of translating G into assembly code can be generalised by using the concept of a *machine description file*. That file contains a mapping from G instructions to a specific machine instruction sequences. A separate program, the *machine description compiler* compiles the machine description file into an executable program that converts G into assembly. This approach has the following advantages:

- The developer need only focus on the assembly mapping of the G instructions when writing the machine description file. Tasks such as the lexical analysis and parsing of the G instructions need to bother him or her.
- A machine description file is smaller and easier to write than a complete translator. Thus it is easy to create a number of machine description files easing the porting of the system.
- Changing the format of the G code (e.g. converting it into a binary stream for efficiency reasons or even directly connecting the two processes) will not invalidate the machine description files and the effort put into them. Simply a new machine description file translator needs to be written.

The format of the machine description file was designed to be the following:

Comment lines are blank lines, or lines containing a hash character.

Header inclusion starts with the symbol `%%{`. All code from that point up to the matching `%}` is copied verbatim to the assembly file. The purpose of this section is to include assembler constant definitions, jump tables, macros etc.

Assembly comment definition is given by the sequence `%%comment`. The character following the comment word is taken to be the assembler comment character. When code for the translator is generated all comments in the assembly file will be removed. This allows the user to put arbitrary comments inside assembly sequences without affecting the efficiency of the translation process (one should remember that a particular instruction sequence can be repeated tens of times on the output). The comment facility of the definition language is not used, because it is quite probable that the comment character of the definition language (the hash) will serve some other purpose in some assembler.

G instruction definitions are introduced by the `%keyword` sequence where `keyword` is the name of a G instructions. An optional set of parameters can be given. The text starting at the following line up to the first `%}` will be generated for that instruction by the translator. Any parameters will be substituted at translation time with the actual parameters following the G instruction. Any local assembly comments are removed at compile time.

The "machine description" to "G to assembly translator", compiler is implemented as a script written in *Perl*. Initially it used to output *lex* code. This took excessively long to compile and the scanners produced were very slow. A speedup to a factor of 7 was achieved by replacing the *lex* output by a C program based on a perfectly hashed function. An option exists to allow the translator to put into the assembly comments indicating the G file and line number that generated each assembly sequence. This is to aid the debugging process.

5.3 Machine Models

A full machine description file for the Motorola 68020 processor has been written. and the machine model for the Intel iAPX386 has been designed.

5.3.1 Motorola 68020

Registers are used on the 68020 [TS86] as follows:

a7 Machine user stack pointer.

a6 C frame pointer

a5 G stack pointer. Always points to the top of stack element.

a4 G heap pointer. Always points to the first free cell.

d0 C function result return.

All other registers can be used for temporary values.

5.3.2 Intel iAPX386

Registers are used on the iAPX386 [SJ87] as follows:

ESP Machine user stack pointer.

EBP C frame pointer

ESI G stack pointer. Always points to the top of stack element.

EDI G heap pointer. Always points to the first free cell.

EAX C function result return.

EBX Secondary heap indexing.

ECX Temporary.

EDX Temporary.

The direction flag must always be set, as string instructions will be used for creating heap cells.

5.4 G implementation

Most of the instruction sequences follow the general pattern presented in [Jon87]. *Unwind* and *eval* are handled through indirect indexed jumps on tables containing the addresses of code for the relevant cell contents. The indexed register indirect with offset addressing mode has been extensively used to compute addresses on the heap in one CPU instruction. For example in the *unwind* implementation putting the index of the net element into the appropriate index of the vertebra (indexed by d1) is accomplished by:

```
movl    a5@(4,d1:l:4),a2              | Get vertebra
movl    a2@(8),a5@(0,d1:l:4)          | Put it on the stack
```

Nodes are built on the hap using the postincrement addressing mode. The `alloc` G instruction uses the 68000 `dbeq` instruction to combine the decrementing of the index, testing and jumping in one instruction. This makes heap allocations very fast.

A number of G primitives have been implemented in assembly. Some more complicated such as `unpack_sum` and `unpack_prod` have been written as recursive functions in C . A C structure and enumeration type have been written to reflect the underlying assembly language organisation. The C function is passed the arguments on the C stack (including the heap pointer). It then assigns the heap pointer to a global variable. C functions that build objects on the heap increment the heap pointer (the heap pointer is declared as a point to a structure of type *cell* and thus is automatically incremented by the correct amount.) At the end of the

computation the glue C function passes the new heap pointer back to the assembly caller. The assembly caller assigns the updated heap pointer to the register variable containing it.

5.5 Cell Implementation

Cells are implemented in the *boxed way* [Jon87, p. 335] The main reason for this is portability. Unboxed representation mixed the pointer bit with the actual value. The resulting code is non-portable. Since some of the runtime environment is written in C (approximately 20%) it was decided to opt for the boxed representation.

Each cell contains a tag field. The tag field is a number that represents how the rest of the cell is to be interpreted. It occupies one 32 bit machine word. Following the tag field are two other fields which contain various values depending on the cell type. Constructors are represented by a linked list of `CONSTR` cells. Each *CONSTR* cell contains a pointer to the next cell and a pointer to its element. Although this implementation is not the most efficient possible (see [JSXX, p. 635] for alternative representation) it easy very general and clean.

In the design of [Jon87] each functions is represented by a different tag and thus at the time of evaluating it its arity can be deduced by its tag. This scheme, although very generic is very difficult to implement in a one pass translator. A different reason was used. When a `globstart` instruction is given the arity of the function is written in statically allocated memory and a pointer is assigned to that address. When the a `pushglobal` instruction is encountered an assembly instruction is emitted to fetch the arity from that address (using the label which is generated in a known way) and place it in the cell.

5.6 Additional G Instructions

Two additional G instruction beyond those specified in [Jon87] have been added. Pushvglobal and `vglobstart` represent functions with a variable number of arguments. The only such functions are the internally used and generated `pack` functions. The instructions are implemented as the corresponding fixed number of argument instructions, only the arity is not put in the cell for `pushvglobal` and is not put into memory for `vglobstart`. These functions are also represented by a different tag. When such an instruction is evaluated or unwound, not check for the number of arguments is made. By definition the correct number of arguments will be on the stack.

5.7 Runtime Environment

An executable HASKELL program needs to be linked with a runtime library. That library contains the internal and HASKELL primitives, runtime error reporting functions and startup code. The startup code allocates space for the stack and a heap into a big structure containing two arrays (which are used by the C functions to address the G memory space) and then calls the G startup code passing the address of that structure as an argument. The G startup code initialises the heap and G stack pointing registers based on that argument (both start on the same point since they grow on opposite directions) and starts evaluating code on the stack.

Chapter 6

System Development Issues

A number of techniques were used during the implementation of this system in order to automate the work,

6.1 Error message management

Error messages are one of the most important aspects of a user interface. Cryptic errors will confuse a novice user, while overly verbose errors can hide valuable details under their volume. Good error messages are very difficult to create as indicated in [Bro82]. An approach often taken is to index the errors by a code. Each error is printed as a brief message accomanied with the index number. In the documentation, and increasingly on-line, there is a list of errors sorted by their index numbers together with more detailed information. The information provided should give possible reasons for the error occuring and suggested recovery actions [BC89, p. 309].

The features described above present a serious logistic problem. Errors have to be numbered in a coherent way. Every time a new error is added the indices and the documentation have to be updated. If on-line help is available then that needs to be kept in sync as well. An attractive solution to this problem is presented in [Dou90].

Errors are kept in a database file. The cause of the error and the proposed solutions are part of the source code. A special program scans the source code and automatically recreates the database. The map between the database and the actual error message is represented by the file name and line number in which the error was called. Special macros are used to take advantage of the C preprocessor ability to substitute the file name and line number for the indentifiers `__FILE__` and

`__LINE__`.

The approach described was modified to acter for the complexity of the HASKELL system. Four classes of errors are distinguished:

Fatal errors These are non-recoverable errors. The system terminates when such an error occurs.

Errors Recoverable errors. The system attempts to recover from the error and continue the opration.

Warnings Diagnostic information. These can indicate something of interest o the user (such as a portability problem) which will not affect the operation of the system.

Runtime errors These are non-recoverable errors found by the runtime system when executing G code. They result in an error message together with the offending address being printed and a temination with a core dump.

Errors are numbered in increments of 1000 for each class. The Perl [Wal88] program `errordb.pl` scans all the source code and creates three files.

`Errors.db` is a file containing the file, line number and error number of each error. It is used by the error reporting functions for associate an error number with a specific error.

`Errors.txt` contains a sorted list of all errors, their brief messages (suitably parameterised in order to hide the C `printf` output codes), possible causes and suggested actions. The file is formated in a uniform and visually attractive way using the `format` facility of Perl. This file is suitable for printing on a line printer or fo display on a glass tty terminal. It is used by the help system of the interpreter to provide additional information when an error is encountered.

`Errors.tex`] is a file containing the error messages with suitable commands for printing by the LaTeX[Lam85] document preparation system. This file was used in the production of this report.

When an error is called one of the parameters is its `CONTEXT` which is replaced using a set of macros by the file name and line number in which the error is used. The error reporting procedure scans `errors.db` to find the error number assigned by `errordb.pl` to that error and prints it out otgether with the brief message and possible other arguments. (The variable number of arguments facility of C

is used to create the error reporting function as a variadic `printf` like funciton. This allows for informative error messages to be printed in an easy and convenient way.)

Chapter 7

Performance

In order to evaluate the performance of the front-end its operation was profiled using the profiler [Spi89].

Table 7.1 shows the result of scanning, parsing and building the syntax tree of the HASKELL prelude. The program was run on a Compaq Deskpro 386/20e running MS-DOS 3.30 with speed reduced from 20MHz to 8MHz to increase the number of hits and consequently the accuracy of the results. Disk caching was disabled to reflect the situation where the program is executed for the first time. Entries with less than 2The program was compiled using Microsoft C 5.1 under the large model with optimisations turned off (the compiler crashes on some of the code when optimisations are enabled).

The profile reveals several interesting things. The dominant factor of the front end is the parsing and construction of the parse tree (it is done within the `yyparse` function. Almost the same percentage of time is spent in the operating system and the system ROM which means that the program has a significant I/O component. One should take into account that the I/O subsystem was proportionally faster than that of a typical machine, since the CPU was slowed down. A significant I/O component often indicates an efficient program. Here I believe that this is the case.

An unexpected result of the profile was the high percentage of time spent in the `memcpy` routine. Searching though the code I found that the only place where the routine was called was within the *fio* library. The purpose of this library is to minimise character copying, so this result was at least ironical. Closer examination of the code revealed that as a result of the maximum line length allowed by *fio*, the average line length of the prelude and the buffer size chosen approximately half of the characters would get copied. The reason for this is subtle:

Fio specifies that a line can be up to 4096 characters. A buffer twice that size

Area	Time %
yyparse	19.55
DOS	15.04
gettoken	12.78
memcpy	12.03
_chkstk	6.02
read	5.26
_fmalloc	4.51
SYSTEM_ROM	4.51
stab_findadd	3.76
_amalloc	3.76
strncmp	2.26
strcmp	2.26
eatwhite	2.26

Table 7.1: Front end profile

was chosen in order to implement a double buffering scheme. Whenever there are less than 4096 characters in the buffer and a new line is requested *fio* needs to copy the remaining characters to the beginning of the buffer and fill the rest in. As the average prelude line is a lot smaller that 4096 characters (typically less than 80), whenever half of the buffer was read, the other half was copied to the beginning. Hence the large percentage spent copying characters. The `memcpy` time can be made arbitrarily small by increasing the size of the buffer or decreasing the maximum allowed line length. Each such doubling or halving will halve the time spent copying.

The time spent in the lexical analyser engine `gettoken` is very respectable compared to the rest. it indicates a sound design and implementation.

The time spent allocating memory is relatively low, as expected given the care taken to avoid allocation of small items.

Surprisingly little time was spent in the string comparison routine given the fact that it is called for every keyword. This dissuaded me from a plan of optimising the keyword recognition scheme.

Chapter 8

Conclusions

This project has demonstrated that the HASKELL language can be implemented. The *whole* language is scanned and parsed and most of its constructs can be intepreted and transformed into efficient machine language.

A significant and ironic aspect of the project is the extensive use of little languages. I do not believe that the 12700 lines of code could have been produced with the reliability achieved were there not for the additional level of abstraction provided by those new languages introduced. HASKELL is a big language. Its many features make it difficult to analyse, parse and reason about. While writing test programs I often found myself wondering what a construct would mean. Having achieved most of the goals set at the beginning of the project I am not convinced that HASKELL provides the linguistic vehicle needed by the functional programming community.

Appendix A

Acknowledgements

A great number of people have helped to make this project a reality. I would like to thank

Sophia Drossopoulou for her insightful comments, Dave Edmondson for many useful ideas and interesting distractions, Sue Eisenbach for having the patience to deal with me, lively discussions and support, Tony Field for his comments on the HASKELL implementation problems, Filippos Frangoulis for his help in the early stages of the project, Tassos Hadjikokolis for his invaluable help in implementing the G primitives and his patience on keeping an eye while I was coding them in assembly, Chris Hankin for his general guidance, Andrew Hume for his expert advice on the implementation of the *fio* library, Natalia Karapanagioti for filling the cold Sun Lounge with her laughter, Paul Kelly for the provision of the December HASKELL report, Lee McLaughlin for providing me with the Glasgow implementation of HASKELL and for the hard work put (together with Stuart McRobert) behind the *UKUUG* software archives—the source of most of the tools used in this project—, LH Software Ltd. for providing me with a C compiler, Keith Septhon for his expert opinion on tricky parts of HASKELL , Jan Simon Pendry for sharing his experience in an attempt to implement the *fio* library, Periklis Tsahageas for various long theoretical and philisophical discussions, down to earth advice and driving me home, Larry Wall for writing Perl—the most used tool after the C compiler—, and finally, various contributors of the project GNU whose tools I have used.

The implementation of this system would not have been possible without the reliabity, speed and disk space of my COMPAQ DESKPRO 386/20e machine. Each one of the 30,100,780,000,000 cycles it went through, from the beginning of the coding till now (the last night before handing in the report) helped the realisation

of the system described.

Appendix B

Error Messages

3001 Error count exceeds *number*; stoping

Reason Too many errors were output.

Action Correct some of the errors and retry.

3002 Out of memory (malloc *number*)

Reason All the memory resources of the system were exhausted.

Action Try to run the system with fewer processes, or increase the swap area. Alternatively simplify the program.

3003 Out of memory (realloc *number*)

Reason All the memory resources of the system were exhausted.

Action Try to run the system with fewer processes, or increase the swap area. Alternatively simplify the program.

2001 Undeclared variable (*string*)

Reason An undeclared variable was used in an expression.

Action Make sure that all variables used are declared.

2002 Pattern match error

Reason A pattern matching failed completely either for different expressions, or cases.

Action Make sure that there is a pattern specification for every different pattern that can exist at runtime.

2003 Single digit expected (*number*)

Reason An invalid fixity declaration was found.

Action Fixity declarations should consist of a single digit in the range from 0 to 9.

2004 Missing constructors in data declaration

Reason No constructors were given in a data declaration of an implementation module.

Action Ensure that the simple type is followed by an equal sign and a constructor list. Constructors can only be ommited in interface modules.

2005 Invalid unary operator'*string*' in expression

Reason A unary operator other than unary minus was found in an expression.

Action Haskell only allows '-' to be used as a unary operator. Ensure brackets are used correctly to indicate precedence.

2006 Invalid successor pattern

Reason A successor pattern composed by invalid elements was found.

Action Ensure that the pattern is composed by a variable to which an integer is added. Check that precedence rules guarantee the correct binding.

2007 Invalid unary operator'*string*' in pattern

Reason A unary operator other than unary minus was found in a pattern.

Action Haskell only allows '-' to be used as a unary operator. Ensure brackets are used correctly to indicate precedence.

2008 Syntax error

Reason A syntactical error was found in the program source.

Action Make sure that the program follows the syntax rules of Haskell. Check for misspellings. Check the input against the precedence rules.

2009 Invalid interface import

Reason An invalid import declaration was found in an interface module.

Action Ensure that no “hiding” declaration has been used and that and explicit import list has been given.

2010 Invalid interface declaration

Reason An invalid variable or class declaration was found in an interface module.

Action Ensure that no “default” declaration has been used and that any variable or class instance declarations only consist of type signatures.

3004 End of file in line comment

Reason The file ended while processing a single line comment.

Action Make sure that the line ends with a newline.

2011 Floating point number overflow

Reason A floating point number with a value higher or lower than the maximum or minimum representable value on this system was encountered.

Action Make sure that the exponent is within the valid range and that no other numbers follow it.

2012 Invalid backquote operator

Reason The operator formed using backquotes was not valid.

Action Ensure that the operator is enclosed within backquotes and that only valid alphanumeric characters are used. No spaces are allowed within the backquotes.

2013 Close bracket missing in variable or constructor

Reason An attempt was made to use a symbolic operator as a curried variable or constructor by encolosing it in parentheses. No closing parenthesis was found.

Action Ensure that the symbols are enclosed within parentheses and that only valid symbol characters are used. No spaces are allowed within the parentheses.

2014 Invalid decimal escape (*number*)

Reason An illegal decimal escape sequence was found. The value of the resulting character is higher than the maximum allowed.

Action Give an decimal number from 0 to 255 and make sure that a sequence of less than three decimal digits is not followed by other digits producing a spurious result.

2015 Invalid octal escape (*octal number*)

Reason An illegal octal escape sequence was found. The value of the resulting character is higher than the maximum allowed or no octal digits were following the ø.

Action Give an octal number from 0 to 377 and make sure that a sequence of less than three octal digits is not followed by other digits producing a spurious result.

2016 Invalid hexadecimal escape (*hexadecimal number*)

Reason An illegal hexadecimal escape sequence was found. The value of the resulting character is higher than the maximum allowed or no hexadecimal digits were following the
x sequence.

Action Give an hexadecimal number from 0 to ff and make sure that a sequence of hexadecimal digits is not followed by other hexadecimal digits producing a spurious result.

2017 Invalid ASCII control escape (*character*)

Reason An ASCII control escape sequence with an invalid control character was detected.

Action Ensure the an uppercase character from A to Z

2018 Invalid backslash escape (*string*)

Reason An unknown backslash escape was found.

Action Make sure that the backslash escape is valid. Valid backslash escapes are a backslash followed by one of a b f n r t v " ' & or the name of an ASCII control, hat followed by an uppercase letter, x followed by a hexadecimal number or an optional o followed by an octal number.

2019 Empty character

Reason A character with no contents was found.

Action Make sure that there is exactly one character between the single quote marks. A character consisting of a single quote mark should be given as ''.

2020 Invalid null character constant

Reason The null character escape & is only meaningful inside string constants.

Action Either denote a string constant using double quotes or change the escape sequence to a valid one.

2021 Invalid character constant ('*character*' 0*octal number number* 0x*hexadecimal number*)

Reason An invalid character constant was given. The constant is not in the ASCII range 32-126.

Action Make sure that the character is withing the ASCII range 32-126. All other characters should be given by backslash escapes.

2022 Possibly unterminated character constant

2023 Newline in string

Reason A newline was detected within a string. A string was not terminated on the same line where it started.

Action Make sure that the string terminates on the same line it starts. To include a newline in a string use the
n escape sequence.

3005 End of file in string

Reason The file ended while reading a string.

Action Make sure that the string ends with a double quote.

2024 Badly formed string gap

Reason A string gap was badly formed.

Action Make sure that the string gap is composed by a backslash, possibly followed by spaces or tabs, followed by a single newline, possibly followed by another series of spaces or tabs, followed by a backslash.

2025 Invalid character in string (’*character*’ 0*octal number number* 0x*hexadecimal number*)

Reason A non printable character was found within a string.

Action Make sure that the string only contains ASCII characters between 32 and 126. All other characters should be given as backslash escapes.

3006 End of file in comment

Reason The file ended while processing a block comment.

Action Make sure that every comment ends with a *minus brace* and that no nested comments have been left open.

2026 Bracket missmatch inside qualifier

Reason A bracket mismatch has been detected while parsing a list comprehension.

Action Make sure that the round, square and curly brackets in the list comprehension are correctly balanced. Brackets of different kinds should never overlap.

3007 Qualifier ring buffer full

Reason Too many lexical tokens while scanning ahead a qualifier were found.

Action Make sure that the square brackets around the list comprehension are correctly balanced. If the expression is very complex try simplifying it.

3008 Premature EOF

Reason End of file was reached while scanning a list comprehension.

Action Make sure that the square brackets around the list comprehension are correctly balanced.

3009 Qualifier ring buffer full

Reason Too many lexical tokens while scanning ahead a qualifier were found.

Action Make sure that the square brackets around the list comprehension are correctly balanced. If the expression is very complex try simplifying it.

3010 Token ring buffer full

Reason

Action Too many lexical tokens were pushed by the layout rules. Try simplifying the layout of the source.

3011 Indentation stack overflow

Reason

Action Too many lexical tokens were pushed by the layout rules. Try simplifying the layout of the source or use some explicit curly brackets.

3012 Indentation stack underflow

Reason

Action Tried to add more implicit close braces than implicit open bracess had been addedusing the layout rules. Curly brackets that have been opened by the user must be explicitly closed.

3013 End of file in line comment

Reason The file ended while processing a single line comment.

Action Make sure that the line ends with a newline.

3014 Unable to open *string*: *string*

Reason The input file specified could not be opened for reading.

Action Make sure that the correct filename and extension were given and that the file has appropriate permissions.

Appendix C

A General Critique of the Haskell Syntax

C.1 Introduction

The syntactic description of Haskell has a number of problems. They can be classified into three different categories of severity. I will analyse these categories from the least problematic to the most troublesome. While all of the problems can in theory be solved, I believe that the implementation of the language will suffer death by a thousand cuts.

C.1.1 Stylistic problems

The easiest problems to solve are the ones associated with the way the language is presented. The syntax given in appendix B of the language specification is supposed to follow some notational conventions which are given in section 1 of the appendix. However in the presentation of the grammar in section 4 a number of additional description mechanisms are used. These are:

- Bracketed comments on the right of the rules. The comments specify anything from the number of times a construct may repeat to the type an operand is allowed to have. I stress that the comments do not have just semantic meaning. A number of them are used to distinguish between different syntactic choices.
- The ellipsis construct is introduced to signify repetition.
- Some productions are subsets of other productions.

While most of these problems can be dealt with they show a certain lack of experience or care from the persons who designed the language. They furthermore demonstrate that the grammar is difficult to describe in a formal way and thus probably also difficult for people to understand, implement and use.

C.1.2 Lexical ties

There is no clear distinction between the lexical and grammatical elements of the language definition. The same categories of lexical elements are used as different types of tokens by the grammar. Thus the lexical analyser needs to have a number of special hooks to change its behaviour according to the context of the syntax. The cases identified up to now are:

- Constant identifiers can serve as constant identifiers, as class specifiers and as constant type specifiers.
- Variable identifiers can serve as variable identifiers and as variable type identifiers.
- Operands can change their precedence and associativity dynamically.
- Layout is significant. Although one might get the impression than the lexical analyser only needs to check white space at the beginning of each line to implement the layout rules a closer examination of the rule reveals that layout can begin at any point within a line.

Most of the problems described above can be found in other languages. C requires a single lexical tie for type definitions [KR78] , *awk* [AKW79] doesn't require statement terminators (one form of the Haskell layout rule), *Ratfor* implements an extended layout rule, Prolog has varying precedence of operators. No language that I know of combines them all together. In some cases even the designers of such solutions regretted their decision because of the problems it added to the language specification and implementation. I believe that most of the above bugs/features were added by a combination of creeping featurism and lack of attention to their implications. They increase the complexity of the language in a number of ways and add confusion to an already complex language.

C.2 Language ambiguity and lack of LR(k)ness

There are many ambiguities in the language. Some of rules for resolving the ambiguities are given in the analytical description of the language, others should

be deduced from the as yet unpublished standard prelude. Yet the fact that 33 disambiguating rules were needed to resolve the ambiguities in a yacc description of the language means that it will be very difficult to guarantee what language will be recognised by the parser. In particular the function application production although ambiguous has no operand associated with it. Thus in a yacc implementation disambiguating rules have to be specified for every language token or literal that might be used to introduce a new expression.

In addition to the ambiguities presented above the language described in the appendix turns out not to be LR(k). Qualifiers can start with either a pattern or an expression and both patterns and expressions can be arbitrarily long lists of variables. Thus an arbitrary number of tokens needs to be scanned before the list can be reduced to a pattern or an expression. This problem can be solved by merging patterns and expressions together. This however places an extremely high burden on the semantic analysis phase which would have to distinguish between the two and verify that no illegal patterns or expressions were entered. As patterns and expressions differ in a number of ways this method introduces a number of problems.

C.3 Conclusions

The syntactical description of the Haskell programming language is from a number of different aspects deficient. This makes an implementation of the language very problematic or even impossible. A solution would be a radical redesign of the language. The new language would have to be designed with an eye towards its implementation, avoiding unproven language concepts and irrelevant baroque features. According to the preface of the Haskell report, Haskell is to be a standard functional programming language. Experimentation with language features irrelevant to the field of functional programming should not be part of such a standard.

Appendix D

Error Log

On February 14 when I started testing I decided on a clean room strategy. All errors found during testing would be logged. The resulting log should provide a base for measuring the relibility of the project, relative difficulty of various parts and provide various software metrics.

Wed Feb 14 20:44:52 1990 Lexical analysis File scan.l

one character reserved operators were not treated specially and I relyied on the default rule to match them. Other rules preceded the default rule and they were used.

Thu Feb 15 09:18:44 1990 Lexical analysis File scan.l

Some one character reserved operators were treated as variable operators. The problem was finaly traced to a - appearing in a character class. This was correctly interpreted by lex as a range, whereas I mean it as a literal.

Sat May 19 16:56:22 1990 Lexical analysis File scan.l

In routine process_string when a backslash escape was found a check for a string gap was made. If that test failed process_escape(_ was called. I forgot to put the character back into the stream.

Sat May 19 17:19:01 1990 Lexical analysis File scan.l

Escape sequences were parsed wrongly. I had used a line of the form:

```
while (c = input() && isdigit(c))
```

which was parsed as:

```
while (c = (input() & isdigit(c)))
```

Sat May 19 17:29:56 1990 Lexical analysis File scan.l

In transllating hexadecimal escape sequences into a number I forgot that c - '0' did not produce the right result for hex. I did not even check of upper and lowercase hex escapes.

Sat May 19 17:36:04 1990 Lexical analysis File scan.l

The conversion from hex character to hex value was done by finding the index of the character into a table. Instead of subtracting the table from the index I subtracted the index from the table.

Sat May 19 17:48:57 1990 Lexical analysis File scan.l

Did not handle the "consume longest lexeme" rule between the ASCII escapes SOH and SO.

Sat May 19 18:03:37 1990 Lexical analysis File scan.l

The code for sting gaps did not take into account that the whitespace before and after the newline was optional.

Sat May 19 18:41:58 1990 Lexical analysis File scan.l

Had forgotten to include space in the characters that are allowed in strings.

Sun May 20 17:08:58 1990 Lexical analysis File scan.c

The get next character macro returned 0 if a character was put back into the stream. I was zeroing the memory to indicate that the pushed back character was taken away and then was returning the same memory. A temporary variable was forseen and initialised to the value of the memory before it was zeroed, but was not returned.

Sun May 20 17:13:12 1990 Lexical analysis File scan.c

Forgotten to put a case for EOF in the switch statement. EOF was passed to the parser as -1 whereas the parser expects 0.

Sun May 20 17:20:16 1990 Lexical analysis File scan.c

Used the operator sizeof within a macro to instead of strlen for efficiency, so that it would be calculated at compile time. Did not think that it returns one more than strlen.

Sun May 20 17:25:57 1990 Lexical analysis File ctype.ctc

Had not included the character at in the list of allowed escapes

Sun May 20 17:58:05 1990 Lexical analysis File ctc.bat

The ctype compiler generates two macros that do nothing, isnone and isany.

These macros discard their arguments and thus did not produce the side effect needed by scan.

Sun May 20 18:17:32 1990 Lexical analysis File scan.c
The (-) varop did not work. When save was called to put it in the symbol table c did not contain '-' as was required by save.

Sun May 20 18:23:48 1990 Lexical analysis File scan.c
Did not put the character after the open bracket taken to examine if it meant the start of a varop back to the input stream if the examination proved negative.

Sun May 20 18:36:33 1990 Lexical analysis File scan.c
Did not parse the sequence (- foo) as four tokens. Attempted to parse it as a varid. Modified for the special case of a -. Not sure if I need to handle any. If so then lookahead is needed. Arghh!!!

Sun May 20 18:36:33 1990 Lexical analysis File scan.c
The test for conop needed to unput one character and set c to :.

Sun May 20 18:47:08 1990 Lexical analysis File scan.c

The hypothesis of Sun May 20 18:36:33 1990 was indeed true. Open bracket symbol does not imply tVARID. The prelude contained (backslash x if x ...). Added lookahead. Did not fix tCONID in the same way. Can there be a case (:+ something)?

Sun May 20 20:57:56 1990 Lexical analysis File scan.c
The layout rules used recursive calls to yylex in order to get the next token. The recursive call invoked the layout rules recursively and resulted in tokens being pushed by reverse order into the ring buffer!

Sun May 20 22:17:13 1990 Lexical analysis File scan.c
When the ring buffer is empty, topyylex returns yylex. Yylex may fill the ring buffer, but it is too late. I will try to fix it by a hack in topyylex (call yylex and then compate the ring?). I feel it is a hack.

Sun May 20 22:36:55 1990 Lexical analysis File scan.c
The fix Sun May 20 22:17:13 1990 did not work. A detailed analysis showed that the correct operation was to call yylex and then swap the token returned with the token in the ring, returning the token in the ring.

Sun May 20 22:42:53 1990 Lexical analysis File scan.c

A very complicated attempt at optimising the copying of yylval in topyylex was wrong. The idea was not to copy yylex when the token has value less than 255. When that involved swapping the two tokens the sequence got very complicated because depending on their values sometimes yylval became clobbered and sometimes not. The correct series of tests is probably more expensive than the actual copy, so the optimisation was removed.

Sun May 20 22:52:43 1990 Lexical analysis File scan.c

Topyylex is still wrong. As done the ring will never empty because it always gets another token before examining the ring. Modified to get another token only when the ring is empty. Afterwards it checks if a sidefect of getting the token was filling the ring and performs the swap operation or returns the token.

Sun May 20 23:03:50 1990 Lexical analysis File scan.c

The recursive calls to yylex and the filling of the ring are not working correctly. The approach of using the lexical analyser to eat white space when looking for the next lexeme is probably not the best. Trying new approach to have a mini scanner to eat white space.

Sun May 20 23:25:38 1990 Lexical analysis File scan.c

It may be that some of the problems between Sun May 20 20:57:56 1990

and Sun May 20 23:03:50 1990 may have been imeagined bacause of wrong input data. The example used to test layout had been typed in long time ago and looked like the example in the April Haskell report. In fact it had many subtle differences which made the scanner look buggy. (When looking at the scanner output I was reading against the Haskell document, not the actual test data.) Closer examination revealed that the example had changed between the September report and the April one!

Sun May 20 23:44:51 1990 Lexical analysis File scan.c

Forgot to take into account the fact that a single line may terminate a number of braces. This is expressed as a while loop, but because yylex is called I have too keep state and use a goto unindent. Argh!!!

Sun May 20 23:54:09 1990 Lexical analysis File scan.c

Initialising the column where module appear to -1 when it appears it too late

because a semicolon whil alread have been inserted when the newline comes. (if there is a newline before module). Fixed by initialising the indentstack to have a -1 entry in it. Also forgot to update the state on EOF. Thus the program was looping outputing semicolons.

Mon May 21 00:51:37 1990 Lexical analysis File scan.c
Did not update state after outputing a ';'. This was the real reason for the looping.

Wed May 30 15:26:02 1990 Parsing File makepr.pl
The procedure that prints out structures checked to se if the pointer to the structure was null, but did not return if it was. Instead it could call itsself recursively.

Wed May 30 18:38:39 1990 Error reporting File error.c
The arguments passed to sscanf were not passed by reference.

Wed May 30 18:39:34 1990 Error reporting File error.c
A brace was missing form a nested if. Consequently the wrong else was chosen.

Wed May 30 18:40:10 1990 Testing File testparse.c
The symbol table was not initialised.

Thu May 31 10:11:00 1990 Parsing File parse.y
The C code for module did not set the dollar dollar variable. This was then set by default to NULL.

Thu May 31 10:26:10 1990 Lexical analysis File scan.c
Forgot to unput the first non digit character read after an integer.

Thu May 31 18:59:05 GMT 1990 Parsing File fix.c
The test for differentiating between constructor operators and variable operators was wrong. Constructor operators start with upparcase and variable operators with lowercase, not the opposite.

Fri Jun 01 14:12:00 1990 Parsing File fix.c
Had assignments for constructor and variable operators swapped around.

Fri Jun 01 14:36:08 1990 Parsing File fix.c
- does not qualify for a symbol and was thus termed as a constructor.

Fri Jun 01 16:15:25 1990 Parsing File parse.y

After adding the open brace in the module derivation, forgot to remove it from the body derivation.

Fri Jun 01 17:08:42 1990 Error management File error.c

When an error could not be located in the error database the file numebr and file name got corrupted. The error number was passed to printf, although no corresponding

Fri Jun 01 21:55:50 1990 Parsing File parse.y

Data declarations were not parsed because type variables were expected. The lexical analyser did not differentiate type from normal variables and thus a syntax error occurred.

Fri Jun 01 23:43:51 1990 Lexical analysis File scan.c

If a closing brace was needed at end of file the end of file character was not pushed back in the stream.

Fri Jun 01 23:53:41 1990 Lexical analysis File scan.c

After moving to a new file failed to initialise input system.

Sat Jun 02 00:05:40 1990 Lexical analysis File scan.c

Did not initialise indentation for layout rules after the ‘interface’ keyword.

Sat Jun 02 01:06:01 1990 Lexical analysis File scan.c

Did not invoke the layout rule after newlines that terminate line comments (–).

Sat Jun 02 01:39:32 1990 Parsing File parse.y

The rule for patterns expected an explicit ’-’ instead of varopl6 as it is renamed by fixity declarations.

Sat Jun 2 16:02:17 GMT 1990 Lexical analysis File scan.c

Did not initialise the ring buffer.

Sat Jun 2 16:04:05 GMT 1990 Symbol table File symtab.c

The printf specification for creating unique identifiers missed the final d for the

Sat Jun 2 16:26:26 GMT 1990 Lexical analysis File scan.c

Tried to interpret 1..2 as a floating point number.

Tue Jun 5 22:20:57 BST 1990 Symbol table File symval.c parse.y
Did not create symbol values for most symbols. This cause the macro that checked symbol values to crash.

Tue Jun 5 23:30:12 GMT 1990 Lexical analysis File scan.c
Erroneously checkd for the symbol value of converted VAROPS (in brackets). This made it return the tokval of the assigned fixit instead of varop.

Wed Jun 6 22:59:10 BST 1990 Parsing File parse.y
Used wrong tag when initialising class structures.

Fri Jun 8 19:07:17 BST 1990 Interpreter File interp.c
Forgot to add suspensions in function applications Checked the value of a variable in the environment after checking if it was a builtin.

Fri Jun 8 19:22:53 BST 1990 Interpreter File interp.c
Did not evaluate the value of a variable in the environment when it was found.

Fri Jun 8 21:11:45 BST 1990 Interpreter File interp.c
Did not pass the name of the unitialised variable in the error function.

Fri Jun 8 21:19:58 BST 1990 Interpreter File testparse.c
Did not initialise the primitives

Fri Jun 8 21:23:43 BST 1990 Lexical analyser File symval.c
Did not check that tokens whose tokval was

Fri Jun 8 22:26:00 BST 1990 Interpreter File interp.c
Used the wrong variable to get the builting name. Used the result of the environment search (which was NULL) instead of the original expression.

Fri Jun 8 22:31:28 BST 1990 Interpreter File prim.c
The tests used in the assertions to verify that the types passed to primitives were correct used assignment instead of equality test.

Fri Jun 8 22:39:46 BST 1990 Interpreter File prim.c
The non-lazy primitives were not evaluating their arguments.

Sat Jun 9 21:41:27 BST 1990 Parsing File parse.y
Expressions did not allow for constructor operators Forgot to change the '+' for successor patterns to its infixl6 op.

Sat Jun 9 22:35:08 BST 1990 Lexical analysis File scan.c

When checking for keywords and saving them, did not update yycolumn.

Sun Jun 10 15:23:54 BST 1990 Lexical analysis File scan.c

When a tab was found a wrong expression was used to incement the column number.

Appendix E

Trademarks

COMPAQ and COMPAQ 386/20e are trademarks of Compaq Computer Corporation.
Intel, 386 and iAPX386 are trademarks of Intel Corporation.
Microsoft and MS-DOS are trademarks of Microsoft Corporation.
Miranda is a trademark of Research Software Ltd.
PDP-11 and VAX are trademarks of Digital Equipment Corporation.
TeX is a trademark of the American Mathematical Society.
Unix is a registered trademark of AT&T in the USA and other countries.
All other trademarks are property of their respective owners.

Bibliography

[AJ74] A. V. Aho and S. C. Johnson. LR parsing. *Computing Surveys*, 6(2):99–124, June 1974.

[AKW79] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. Awk – a pattern scanning and processing language. *Software – Practice and Experience*, 9(4):267–280, 1979.

[AKW88] Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK Programming Language*. Addison-Wesley, 1988.

[aNFV84] P. W. E. Verhelst an N. F. Verster. PEP: an interactive programming system with an algol-like programming language. *Software—Practice and Experience*, 14:119–133, 1984.

[ASU85] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1985.

[Bai85a] Paul A. Bailes. A low-cost implementaion of coroutines for C. *Software—Practice and Experience*, 15(4):379–395, April 1985.

[Bai85b] D. Bailey. The university of salford Lisp/Prolog system. *Software—Practice and Experience*, 15(6):595–609, June 1985.

[BC89] Judith R. Brown and Steve Cunningham. *Programming the User Inteface*. John Wiley & Sons, 1989.

[Ben82] Jon Louis Bentley. *Writing Efficient Programs*. Prentice-Hall, 1982.

[Ben85] Jon Bentley. Associative arrays. *Communications of the ACM*, 28(6):570–576, June 1985.

[Ben86] Jon Bentley. Little languages. *Communications of the ACM*, 29(8):711–721, August 1986.

[Bou86] S. R. Bourne. An introduction to the unix shell. In *UNIX Users' Supplementary Documents.* Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California 94720, April 1986. 4.3 Berkeley Software Distribution.

[Bro82] P. J. Brown. 'my system gives excellent error messages' — or does it? *Software—Practice and Experience*, 12:91–94, 1982.

[Car85] Luca Cardelli. The amber machine. Technical Report 119, Bell Laboratories, Murray Hill, New Jersey 07974, June 1985.

[CRRS89] Diane Barlow Close, Arnold D. Robbins, Paul H. Rubin, and Richard Stallman. *The GAWK Manual.* Free Software Foundation, 675 Massachusetts Avenue, Cambridge, MA 02139 USA, 0.11 beta edition, October 1989.

[Dav84] Jack W. Davidson. Register allocation and exhaustive peephole optimization. *Software—Practice and Experience*, 14(9):857–865, September 1984.

[DGM80] John M. Dedourek, Uday G. Gujar, and Marion E. McIntyre. Scanner design. *Software—Practice and Experience*, 10:959–972, 1980.

[Don87] Chris S. Mc Donald. fsh — a functional UNIX command interpreter. *Software—Practice and Experience,* 17(10):685–700, October 1987.

[Dou90] Rohan T. Douglas. Error message management. *Dr. Dobb's Journal*, pages 48–51, January 1990.

[Fed84] J. Feder. The evolution of UNIX system performance. *Bell System Technical Journal*, 63(8):1791–1814, October 1984.

[FH82] Christopher W. Fraser and David R. Hanson. Exploiting machine-specific pointer operations in abstract machines. *Software—Practice and Experience*, 12:367–373, 1982.

[FH88] Anthony J. Field and Peter G. Harrison. *Functional Programming.* Addison-Wesley, 1988.

[GH81] Martin L. Griss and Anthony C. Hearn. A portable LISP compiler. *Software—Practice and Experience,* 11:541–605, 1981.

[Gol80] Adele Goldberg. *Smalltalk 80: The Language and its Implementation.* Addison Wesley, 1980.

[Han83] David R. Hanson. Simple code optimizations. *Software—Practice and Experience*, 13:645–763, 1983.

[Heu86] V. P. Heuring. The automatic generation of fast lexical analysers. *Software—Practice and Experience*, 16(9):801–808, September 1986.

[HJ87] Samuel P. Harbison and Guy L. Steele Jr. *C: A Reference Manual.* Prentice–Hall, second edition, 1987.

[HL87] R. Nigel Horspool and Michael R. Levy. Mkscan — an interactive scanner generator. *Software—Practice and Experience*, 17(6):369–378, June 1987.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[HOS85] Christoph M. Hoffmann, Michael J. O'Donnel, and Robert I. Strandh. Implementation of an interpreter for abstract equations. *Software—Practice and Experience*, 15(12):1185–1204, December 1985.

[Hud89] Paul Hudak. Conception, evolution and application of functional programming languages. *Communications of the ACM*, 21(3):359–411, September 1989.

[Hum88] Andrew Hume. Grep wars: The strategic search initiative. In *Proceedings of the EUUG Spring 88 Conference*, pages 237–245. European UNIX User Group, 1988.

[Hum90] Andrew Hume. Private communication of A. Hume of Bell Laboratories, Murray Hill, New Jersey 07974, May 1990.

[HWA+89] Paul Hudak, Philip Wadler, Arvind, Brian Boutel, Jon Fairbairn, Joseph Fasel, John Hughes, Thomas Johnson, Dick Kieburtz, Rishiyur Nikhil, Simon Peyton Jones, Mike Reeve, David Wise, and Jonathan Young. Report on the programming language haskell. Technical Report Version 1.0, Yale University, University of Glasgow, September 1989.

[HWA+90] Paul Hudak, Philip Wadler, Arvind, Brian Boutel, Jon Fairbairn, Joseph Fasel, John Hughes, Thomas Johnson, Dick Kieburtz, Rishiyur

Nikhil, Simon Peyton Jones, Mike Reeve, David Wise, and Jonathan Young. Report on the programming language haskell. Technical Report Version 1.0, Yale University, University of Glasgow, April 1990.

[JJ89] S. L. Peyton Jones and M. S. Joy. FLIC – a funcitonal language intermediate code. Internal Note 2048, University College London, Department of Computer Science, July 1989.

[JL87] S. C. Johnson and M. E. Lesk. Language development tools. *Bell System Technical Journal*, 56(6):2155–2176, July-August 1987.

[Joh75] Stephen C. Johnson. YACC – yet another compiler-compiler. Technical Report 32, Bell Laboratories, Murray Hill, New Jersey 07974, July 1975.

[Joh82] S. C. Johnson. A tour through the portable C compiler. In *UNIX Programmer's manual: Supplementary Documents*, volume 2, pages 544–568. Holt, Rinehart and Winston, seventh edition, 1982.

[Jon86] Simon L. Peyton Jones. Parsing distfix operators. *Communications of the ACM*, 29(2):118–122, February 1986.

[Jon87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[Joy86] W. N. Joy. An introduction to the C shell. In *UNIX User's Supplementary Documents, Volume 1*. Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley, California 94720, April 1986. 4.3 Berkeley Software Distribution.

[JR81] S. C. Johnson and D. M. Ritchie. The C language calling sequence. Technical Report 102, Bell Laboratories, Murray Hill, New Jersey 07974, September 1981.

[JSXX] Guy Lewis Steele Jr. and Gerald Jay Sussman. Design of a LISP-based microprocessor. *Communications of the ACM*, XXX(XXX):628–644, XXX 1980 XXX.

[KC] B. W. Kernighan and L. L. Cherry. A system for typesetting mathematics. Technical Report 17, Bell Laboratories, Murray Hill, New Jersey 07974.

[Ker] Brian W. Kernighan. A typesetter-independent TROFF. Technical Report 97, Bell Laboratories, Murray Hill, New Jersey 07974.

[KKM80] Peter Kornerup, Bent Bruun Kristensen, and Ole Lehrmen Madsen. Interpretation and code generation based on intermediate languages. *Software—Practice and Experience*, 10:635–658, 1980.

[Kli81] Paul Klint. Interpretation techniques. *Software—Practice and Experience*, 11:963–973, 1981.

[KR78] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice-Hall, first edition, 1978.

[KR82] B. W. Kernighan and D. M. Ritchie. UNIX programming. In *UNIX Programmer's manual: Supplementary Documents*, volume 2, pages 301–322. Holt, Rinehart and Winston, seventh edition, 1982.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, second edition, 1988.

[Lam81] David Alex Lamb. Construction of a peephole optimizer. *Software—Practice and Experience*, 11:639–647, 1981.

[Lam85] Leslie Lamport. *LATEX: A Document Preparation System*. Addisson-Wesley, 1985.

[Lan63] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6:308–320, 1963.

[Lei84] Philip Leith. Top-down design within a functional environment. *Software—Practice and Experience*, 14(10):921–930, October 1984.

[Les75] M. E. Lesk. Lex – a lexical analyzer generator. Technical Report 39, Bell Laboratories, Murray Hill, New Jersey 07974, October 1975.

[Les82] M. E. Lesk. TBL – a program to format tables. In *UNIX Programmer's manual: Supplementary Documents*, volume 2, pages 157–174. Holt, Rinehart and Winston, seventh edition, 1982.

[Lic86] Zavdi L. Lichtman. The function of T and NIL in LISP. *Software—Practice and Experience*, 16(1):1–3, January 1986.

[LPT82] Olivier Lecarme, Mireille Pellissier, and Marie-Claude Thomas. Computer-aided production of language implementation systems: A review and classification. *Software—Practice and Experience*, 12:785–824, 1982.

[LV73] D. Lurié and C. Vandoni. Statistics for FORTRAN identifiers and scatter storage techniques. *Software—Practice and Experience*, 3:171–177, 1973.

[Mar84] B. L. Marks. Taming the PL/1 syntax. *Software—Practice and Experience*, 14(8):775–789, August 1984.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(12):184–195, December 1960. Part I.

[MSQ88] Microsoft Corporation, 16011 NE 36th Way, Box 97017, Redmon, WA 98073-9717. *Microsoft QuickBASIC*, 4.5 edition, 1988.

[Nai84] Lee Naish. *Mu-prolog 3.1db Reference Manual*. Melbourne University, 1984.

[Oss82] J. F. Ossanna. NROFF/TROFF user's manual. In *UNIX Programmer's manual: Supplementary Documents*, volume 2, pages 196–229. Holt, Rinehart and Winston, seventh edition, 1982.

[Pal82] Jacob Palme. Uses of the SIMULA process concept. *Software—Practice and Experience*, 12:153–161, 1982.

[Pax89] Vern Paxson. *flex: fast lexical analyzer generator*. Real Time Systems, Bldg, 46A, Lawrence Berkeley Laboratory, Berkeley CA 94720, 1989. Optional.

[Per87] William E. Perry. *A Standard for Testing Application Software*. Auerbach, Boston, MA, 1987.

[PFTV88] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, 1988.

[Rob83] D. J. Robson. An evaluation of throw-away compiling. *Software—Practice and Experience*, 13:241–249, 1983.

[San78] Erik Sandewall. Programming in an interactive environment: the "LISP" experience. *Computing Surveys*, 10(1):35–71, March 1978.

[Set84] Ravi Sethi. Preprocessing embedded actions. *Software—Practice and Experience*, 14(3):291–297, March 1984.

[SJ87] Bud E. Smith and Mark T. Johnson. *Programming the Intel 80386.* Scott, Foresman and Company, 1987.

[Spi89] Diomidis Spinellis. v08i002: A c execution profiler for ms-dos. Posted in the Usenet newsgroup comp.sources.misc, August 1989. Message-ID: <64297@uunet.UU.NET>.

[Sta84] R. M. Stallman. EMACS: The extensible, customizable, self-documenting display editor. In D. R. Barstow, H. E. Shrobe, and E. Sandwell, editors, *Interactive Programming Environments*, pages 300–325. McGraw-Hill, 1984.

[Sta89] Richard M. Stallman. The GNU C compiler. Distributed by the Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, July 1989.

[Str77] George O. Strawn. Does APL really neeed run-time parsing. *Software—Practice and Experience*, 7:193–200, 1977.

[Suz82] Norihisa Suzuki. Analysis of pointer "rotation". *Communications of the ACM*, 25(5):330–335, May 1982.

[Tei78] Warren et al Teitelman. *InterLISP Reference Manual.* Xerox Palo Alto Research Center, Palo Alto, California, third revision edition, 1978.

[TS86] Walter A. Triebel and Avtar Singh. *The 68000 Microprocessor: Architecture, Software and Interfacing Techniques.* Prenice Hall, 1986.

[Tur79] D. A. Turner. A new implementation technique for applicative languages. *Software—Practice and Experience*, 9(1):31–49, January 1979.

[V7.82] *UNIX Programmer's manual*, volume 1. Holt, Rinehart and Winston, seventh edition, 1982.

[V885] AT&T Bell Laboratories, Murray Hill, New Jersey. *UNIX Time-Sharing System, Programmer's Manual, Research Version*, February 1985. Eighth Edition.

[Wai85] W. M. Waite. Treatment of tab characters by a compiler. *Software—Practice and Experience*, 15(11):1121–1123, November 1985.

[Wai86] W. M. Waite. The cost of lexical analysis. *Software—Practice and Experience*, 16(5):473–488, May 1986.

[Wal88] Larry Wall. *Perl – Practical Extraction and Report Language*, March 1988.

[War83] David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, Artificial Intelligence Center, Computer Science and Technology Division, 333 Ravenswood Ave., Menlo Park, CA 94025, October 1983.

[Wir77] N. Wirth. Design and implemenation of modula. *Software—Practice and Experience*, 7:67–84, 1977.

[Wir85] Niklaus Wirth. *Programming in Modula-2.* Springer Verlag, third edition, 1985.

[Wyk86] Christopher J. Van Wyk. AWK as glue for programs. *Software—Practice and Experience*, 16(4):369–388, April 1986.