University of Edinburgh
Division of Informatics

Translating A Linear Functional Programming Language into C

4th Year Project Report
Computer Science

Nicholas Brown

June 6, 2000

Abstract: We show how a compiler that translates a linearly typed first order functional language into malloc()-free C code can be implemented. The language that the compiler implements gives the user explicit control over heap space usage by introducing a resource type, .
In realizing this language and translating it to into C code, the compiler implements a few novel features. To insure linearity the compiler must check that typing contexts are disjoint. This was implemented by introducing a sets data type, and maintaining sets of heap types used within a context, thus allowing for efficient testing of disjointness. As C does not have a let construct to allow scoping to be introduced, it is necessary to produce many distinct function-global variables to hold intermediate results.
Although other state-of-the-art optimising compilers for functional languages such as OCAML [7] can offer very efficient space usage the advantage of the compiler developed here is that it allows reasoning about about space usage and specifically allow a guaranteed bound on the heap space that will be used. This has mean that the compiler(with further work) will have possible applications in environments where memory is premium, such as programming for embedded systems or computation over the Internet.

Acknowledgements

I would like to thank my Supervisor, Martin Hofmann, for proposing this project, all his help in implementing the compiler and correcting bugs within the code.

Contents

1. Introduction

This paper is primarily concerned with implementing a concrete example of a compiler for a functional programming language that has an affine linear type system and an explicit heap resource type. Such a compiler would produce code that always modifies it's heap allocated arguments in place, thus guaranteeing that an upper bound can be placed on the amount of heap spaced used.

1.1 Motivation

Writing code which manipulates heap allocated data structures such as lists, trees and queues, in-place using a traditional imperative language like C is often awkward and can frequently lead to errors. As an example consider writing a function to reverse a list. Given the following definition of a list type,

```
typedef enum {NIL, CONS} kind_t;

typedef struct lnode {
  kind_t kind;
  int hd;
  struct lnode * tl;
} list_t;
```

then writing the function,

```
list_t reverse(list_t l)
```

which reverses a list passed to it without allocating more space, is not particularly straight forward even with the use of recursion. Now, consider the problem of writing the function,

```
list_t sort(list_t l)
```

which sorts a given list in-place using the well known insertion sort algorithm. This function cannot draw on a previously written function,

```
list_t insert(int a, list_t l)
```

that inserts an element into a already sorted list, as this insertion function allocates space for the new element. Thus, what would appear to be a reusable function is not and the wheel must be re-invented. Although we can define functions in C that manipulate their arguments without allocating space, it is often difficult and not particularly intuitive.

If we now turn to a functional language such as OCAML [7], it is much easier to formulate such functions. If we consider the previous example of reversing a list,

```
let reverse l = let rec rev_aux l acc =
  match l with
      [] -> acc
  | a::l -> rev_aux l (a::acc)
in rev_aux l []
```

This function is easily written and is amenable to equational reasoning, thus being verifiable.

However, this function does not modify its arguments in-place. Instead it allocates new heap space in which to place it's result, as do all other functional programs, If its arguments are not need anymore then the heap space they occupy will eventually be reclaimed for future use by the garbage collector. However we do not know when, and indeed if this will happen, and we may run out of heap space when in-fact we should have sufficient available.

Thus a functional program will generally use more space, less efficiently and in an less predictable way than a similar program written in C. Of course the C code must follow the in-place design mentioned above, explicitly managing it heap resource for this to be true.

It would therefore be beneficial if we could have a functional language that guaranteed to modify its arguments in-place, or a method for easily producing C code that modifies it's arguments in-place and was open to verification.

The aim of the this paper is show that such a goal can be achieved. By Implementing a compiler for a linearly typed functional language that is translated into `malloc()`-free C we have system in which we have the benefits of both. The functional language is easy to write and reason about, and as the compiler for this produces code which does not allocate any heap space we can be sure of efficient memory usage. In fact we can place a bound on the amount of memory used.

This system will thus have advantages over traditional systems in situations where space is at premium. With the ability to reason about the space usage of generated code and guarantee a bound on its use, the system has possible application in areas such as programming for embedded systems and computation over the Internet.

1.2 Background

In his paper, [4] Martin Hofmann proposes a type system for a functional programming language with inductive data types and unrestricted recursive operators over these, that insures that all definable first order programs are polynomial time computable. As well as proving the correctness of the type system it is suggested that all functions can also be evaluated in linear space, that is with out dynamic storage allocation.

In his second paper [5], he goes on to develop a 'design pattern' for compiling linearly typed first order functional programs into C code which modifies it heap allocated arguments without pointer surgery. The main result of this paper is the proof of the correctness of this compilation process. He presents examples of hand compiled functions on trees and lists that follow the given rules for translation into C.

Although there other system available that attempt to control heap space usage of functional program(see [5] for a full discussion on such systems) none are able to offer a guarantee on the amount of space used. A similar effect to in-place manipulation can be achieved using a reference counting garbage collector. This works by tracking heap resources and deallocating them as soon as they are no longer needed. If this is used in conjunction with linear arguments then a runtime space usage performance comparable to the proposed system can be achieved. However, again the proposed system has the advantage it can guarantee a bounded heap size. It has the further advantage that this can be accomplished on any system that supports C.

This in-place bounded heap usage is achieved by introducing a special resource type, $\diamond$. This type points to a piece of free heap space of appropriate size to hold the largest basic element of all data types. The basic premise is that when a recursive constructor function is called it must explicitly be passed an enough arguments of this type to allow it allocate the required heap space. In a similar way, when a recursive data type is decomposed these $\diamond$ are free up. By using a linear typing system over this heap resource type it is possible to ensures that the heap is used safely.

This $\diamond$ type can be represent in C by a pointer type, for example `void *` which points to an appropriately sized piece of the heap. This use of an explicit pointer argument will be familiar to most C programmers, however this system allows the programmer to concentrate on the semantics of their code, the compiler deals with the manipulation of these pointer.

As way of an example, for the list type mentioned previously, we can define the following constructor and destructor functions,

```
typedef void * dia_t;
```

and

```
list_t nil(){
  list_t res;
  res.kind=NIL;
  return res;
}
```

and

```
list_t cons(dia_t d, int hd, list_t tl) {
  list_t res;
  res.kind = CONS;
  res.hd = hd;
  *(list_t *)d = tl;
  res.tl = (list_t *)d;
  return res;
}
```

followed by

```
typedef struct {
  kind_t kind;
  dia_t d;
  int hd;
  list_t tl;
} list_destr_t;
```

and

```
list_destr_t list_destr(list_t l) {
  list_destr_t res;
  res.kind = l.kind;
  if (res.kind == CONS) {
    res.hd = l.hd;
    res.d = (void *) l.tl;
    res.tl = *l.tl;
  }
  return res;
}
```

`dia_t` is a pointer to heap space, representing the type $\diamond$. `nil()` is the obvious empty list constructor. The `cons()` function takes the normal arguments, the list entry `hd` and the list `tl`. However it also takes the afore mentioned pointer to heap space, into which it constructs the new list. Unlike traditional functions, this means that space must be specifically passed to the function, it cannot allocate it's own space. Note the lack of any calls to `malloc()`. The destructor function `list_destr()` return the expected head and tail of a list but also makes available the heap space freed up my this deconstruction of `d`.

It is now straight forward to define a function to reverse a list as mentioned above,

```
list_t rev_aux(list_t l0, list_t acc) {
  list_destr_t l = list_destr(l0);
  return  l.kind==NIL ? acc
   : rev_aux(l.tl, cons(l.d, l.hd, acc));
}

list_t reverse(list_t l) {
  return rev_aux(l,nil());
}
```

It should be noted that this definition does not make calls to `malloc()` so modifies it argument in-place. Yet is still easily understand.

If this method of explicitly passing pointers to heap space is to be used successfully, then functions must be careful not to overwrite memory which is still need. This is achieved my allowing functions to use there heap allocated arguments at most once. This can be implemented using a linear type system.

Instead of relying on a programmer to follow such guidelines, we can introduce an automated system in the form of a compiler that can guarantee the restraints. Further using such a system would free the programmer from re-writing similar code for different data types. For example, if a function was required to reverse a list of pairs instead of integers then the necessary constructor's and types would be needed to be hand produced, but a compiler could produce them directly.

This paper is concerned with implementing such a compiler.

1.3 Synopsis

chapter 1 The introduction, outlining the main goals of the paper, and setting the context for the rest of the work. It gives the motivation for implementing the compiler and presents the previous work upon which the paper is based.

chapter 2 The language for which the compiler is being implemented is introduced. The constructs of the language are presented formally with typing and translation rules given. Unique feature of the language are mentioned.

chapter 3 The design principals followed are explained, as is the structure of the compiler. Each of the stages is explained in turn, outlining the data structures and algorithms used. The methods used to implement the rules given in chapter 2 are discussed.

chapter 4 How the data structures and algorithms outlined in chapter 3 are implemented is explained along with some more concrete design decisions, and coding problems that had to be overcome.

chapter 5 A discussion of how well the compiler implements the language and its usefulness, including how the compiler was tested for errors. The compiler is compared with others for different functional languages, including New Jersey ML and OCAML.

chapter 6 Further work that is need to complete the compiler are mentioned along with ways in which it could be made more efficient and robust. Extension to the language and compiler that would increase there usefulness are also outlined.

chapter 7 The conclusions that can be drawn from the work done in this paper are given, as well as discussing the possible applications for the language and compiler.

2. The Language

We are now ready to formally specify the functional language and how it is translated into C code. By formalising the method outlined in the last chapter and implementing it the usage restrictions upon which the correctness of the method relies can be enforced. A formal proof of the correctness of the translation that will be given below can be found in [5]. It is from the formalisation given below that our compiler specification will be drawn.

The remainder of this chapter is a minor modification of Section 3 of Martin Hofmann's Draft Paper, [5]. Error's in the translation rules have been corrected. Further corrections have been made to the example functions introduced in section 2.3.

2.1 Typing Rules

The zero-order types are given by the following grammar.

$$A ::= \mathsf{N} \mid \diamond \mid \mathsf{L}(A) \mid \mathsf{T}(A) \mid A_1 \otimes A_2$$

More type formers such as sum types, records, and variants can easily be added.

A first-order type is an expression of the form $T = (A_1, \dots, A_n) \to B$ where $A_1 \dots A_n, B$ are zero-order types.

A signature Σ is a partial function from identifiers (thought of as *function symbols*) to first-order types.

A *typing context* Γ is a finite function from identifiers (thought of as parameters) to zero order types; if $x \notin \mathrm{dom}(\Gamma)$ then we write $\Gamma, x{:}A$ for the extension of Γ with $x \mapsto A$. More generally, if $\mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Delta) = \emptyset$ then we write Γ, Δ for the disjoint union of Γ and Δ. If such notation appears in the premise of a rule below it is implicitly understood that these disjointness conditions are met.

Types not including $\mathsf{L}(-), \mathsf{T}(-), \diamond$ are called *heap-free*, e.g. N and $\mathsf{N} \otimes \mathsf{N}$ are heap-free.

Let Σ be a signature. The *typing judgement* $\Gamma \vdash_\Sigma e : A$ read "expression e has type A in typing context Γ and signature Σ" is defined by the following rules.

$$\frac{x \in \mathrm{dom}(\Gamma)}{\Gamma \vdash_\Sigma x : \Gamma(x)} \qquad (\textsc{Var})$$

$$\frac{\Sigma(f) = (A_1, \dots, A_n) \to B \qquad \Gamma_i \vdash_\Sigma e_i : A_i \text{ for } i = 1 \dots n}{\Gamma_1, \dots, \Gamma_n \vdash_\Sigma f(e_1, \dots, e_n) : B} \qquad (\textsc{Sig})$$

$$\frac{\Gamma, x{:}A, y{:}A \vdash_\Sigma e : B \qquad A \text{ heap-free}}{\Gamma, x{:}A \vdash_\Sigma e[x/y] : B} \tag{Contr}$$

$$\frac{c \text{ a } \mathtt{C} \text{ integer constant}}{\Gamma \vdash_\Sigma c : \mathsf{N}} \tag{Const}$$

$$\frac{\Gamma \vdash_\Sigma e_1 : \mathsf{N} \qquad \Delta \vdash_\Sigma e_2 : \mathsf{N} \quad \star \text{ a } \mathtt{C} \text{ infix opn.}}{\Gamma, \Delta \vdash e_1 \star e_2 : \mathsf{N}} \tag{Infix}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathsf{N} \qquad \Delta \vdash_\Sigma e' : A \qquad \Delta \vdash_\Sigma e'' : A}{\Gamma, \Delta \vdash_\Sigma \mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'' : A} \tag{If}$$

$$\frac{\Gamma \vdash_\Sigma e : A \qquad \Delta \vdash_\Sigma e' : B}{\Gamma, \Delta \vdash_\Sigma e \otimes e' : A \otimes B} \tag{Pair}$$

$$\frac{\Gamma \vdash_\Sigma e : A \otimes B \qquad \Delta, x{:}A, y{:}B \vdash_\Sigma e' : C}{\Gamma, \Delta \vdash_\Sigma \mathsf{match}\ e\ \mathsf{with}\ x \otimes y \Rightarrow e' : C} \tag{Split}$$

$$\Gamma \vdash_\Sigma \mathsf{nil}_A : \mathsf{L}(A) \tag{Nil}$$

$$\frac{\Gamma_d \vdash_\Sigma e_d : \diamond \qquad \Gamma_h \vdash_\Sigma e_h : A \qquad \Gamma_t \vdash_\Sigma e_t : \mathsf{L}(A)}{\Gamma_d, \Gamma_h, \Gamma_t \vdash_\Sigma \mathsf{cons}(e_d, e_h, e_t) : \mathsf{L}(A)} \tag{Cons}$$

$$\frac{\begin{array}{l}\Gamma \vdash_\Sigma e : \mathsf{L}(A) \\ \Delta \vdash_\Sigma e_{\mathsf{nil}} : B \\ \Delta, d{:}\diamond, h{:}A, t{:}\mathsf{L}(A) \vdash_\Sigma e_{\mathsf{cons}} : B\end{array}}{\Gamma, \Delta \vdash_\Sigma \mathsf{match}\ e\ \mathsf{with}\ \mathsf{nil} \Rightarrow e_{\mathsf{nil}} | \mathsf{cons}(d, h, t) \Rightarrow e_{\mathsf{cons}} : B} \tag{List-Elim}$$

$$\frac{\Gamma \vdash_\Sigma e : A}{\Gamma \vdash_\Sigma \mathsf{leaf}(e) : \mathsf{T}(A)} \tag{Leaf}$$

$$\frac{\begin{array}{c}\Gamma_{d1} \vdash_\Sigma e_{d1} : \diamond \qquad \Gamma_{d2} \vdash_\Sigma e_{d2} : \diamond \qquad \Gamma_a \vdash_\Sigma e_a : A \\ \Gamma_l \vdash_\Sigma e_l : \mathsf{T}(A) \qquad \Gamma_r \vdash_\Sigma e_r : \mathsf{T}(A)\end{array}}{\Gamma_{d1}, \Gamma_{d2}, \Gamma_a, \Gamma_l, \Gamma_r \vdash_\Sigma \mathsf{node}(e_{d1}, e_{d2}, e_a, e_l, e_r) : \mathsf{T}(A)} \tag{Node}$$

$$\frac{\Gamma \vdash_\Sigma e : \mathsf{T}(A) \qquad \Delta, a{:}A \vdash_\Sigma e_{\mathsf{leaf}} : B \qquad \Delta, d_1{:}\diamond, d_2{:}\diamond, a{:}A, l{:}\mathsf{T}(A), r{:}\mathsf{T}(A) \vdash_\Sigma e_{\mathsf{node}} : B}{\Gamma, \Delta \vdash_\Sigma \mathsf{match}\ e\ \mathsf{with}\ \mathsf{leaf}(a){\Rightarrow}e_{\mathsf{leaf}} | \mathsf{node}(d_1, d_2, a, l, r){\Rightarrow}e_{\mathsf{node}} : B} \quad (\textsc{Tree-Elim})$$

The symbol $\star$ in rule INFIX ranges over a set of binary infix operations such as `+`, `-`, `/` , `*`, `<=`, `==`, ... We may include more such operations and also other base types such as floating point numbers or characters.

As usual, we omit type annotations wherever possible. The constructs involving match bind variables.

Application of function symbols or operations to their operands is linear in the sense that several operands must in general not share common free variables. This is because of the implicit side condition on juxtaposition of contexts mentioned above. In view of rule CONTR, however, variables of a heap-free type may be shared and moreover the same free variable may appear in different branches of a case distinction as follows e.g. from the form of rule IF. Here is how we typecheck $x + x$ when $x{:}\mathsf{N}$. First, we have $x{:}\mathsf{N} \vdash x : \mathsf{N}$ and $y{:}\mathsf{N} \vdash y : \mathsf{N}$ by VAR. Then $x{:}\mathsf{N}, y{:}\mathsf{N} \vdash x{+}y : \mathsf{N}$ by INFIX and finally $x{:}\mathsf{N} \vdash x{+}x : \mathsf{N}$ by rule CONTR. It follows by standard type-theoretic techniques that typechecking for this system is decidable in linear time.

A *program* consists of a signature Σ and for each symbol $f : (A_1, \dots, A_n) {\to} B$ a term

$$x_1{:}A_1, \dots, x_n{:}A_n \vdash_\Sigma e_f : B$$

2.2 Translation into C

Our aim is to compute the meanings of programs by `malloc()`-free `C`-programs.

Let a program $P = (\Sigma, (e_f)_{f \in \mathrm{dom}(\Sigma)})$ be given. Let types(P) be the (finite) set of zero-order types mentioned in P. To each $A \in$ types(P) we assign a unique identifier $\nu(A)$. Next, for each $A \in$ types(P) we introduce a piece of code $[\![A]\!]^{\mathtt{C}}$ defining a `C` type $\nu(A)$ and some functions associated with A.

The code $[\![\diamond]\!]^{\mathtt{C}}$ is: `typedef void * ν(◇);`

The code $[\![\mathsf{N}]\!]^{\mathtt{C}}$ is:

```
typedef int ν(N);
```

The code $[\![A \otimes B]\!]^{\mathtt{C}}$ is:

```
typedef struct {
  ν(A) fst;
  ν(B) snd;
} ν(A⊗B);
```

followed by

```
ν(A⊗B) ν(A⊗B)_pair
(ν(A) fst, ν(B) snd){
  ν(A⊗B) res;
  res.fst = fst;
  res.snd = snd;
  return res;
}
```

The code $[\![\mathsf{L}(A)]\!]^{\mathtt{C}}$ is:

```
typedef struct {
  kind_t kind;
  ν(A) hd;
  void * tl;
} ν(L(A));

ν(L(A)) ν(L(A))_nil(void){
  ν(L(A)) res;
  res.kind = NIL;
  return res;
}

ν(L(A)) ν(L(A))_cons
(ν(◇) d, ν(A) hd, ν(L(A)) tl){
  ν(L(A)) res;
  res.kind = CONS;
  res.hd = hd;
  *(ν(L(A)) *)d = tl;
  res.tl = d;
  return res;
}
```

followed by

```
typedef struct {
  kind_t kind;
  ν(A) hd;
  ν(L(A)) tl;
  ν(◇) d;
} ν(L(A))_destr_t;

ν(L(A))_destr_t ν(L(A))_destr
(ν(L(A)) l){
  ν(L(A))_destr_t res;
  res.kind = l.kind;
  if(res.kind == CONS) {
    res.hd = l.hd;
    res.d = (void *)l.tl;
    res.tl = *l.tl;
  }
  return res;
}
```

The code $[\![\mathsf{T}(A)]\!]^{\mathtt{C}}$ is:

```
typedef struct {
  kind_t kind;
  ν(A) label;
  void * left;
  void * right;
} ν(T(A));

ν(T(A)) ν(T(A))_leaf
(ν(A) label){
  ν(T(A)) res;
  res.kind=LEAF;
  res.label=label;
  return res;
}

ν(T(A)) ν(T(A))_node
(ν(◇) d1, ν(◇) d2, ν(A) label,
    ν(T(A)) left, ν(T(A)) right){
  ν(T(A)) res;
  res.kind = NODE;
  res.label = label;
  *(ν(T(A)) *)d1 = left;
  *(ν(T(A)) *)d2 = right;
  res.left = d1;
  res.right = d2;
  return res;
}
```

followed by

```
typedef struct {
  kind_t kind;
  ν(A) label;
  ν(◇) d1, d2;
  ν(T(A)) left, right; } ν(T(A))_destr_t;

ν(T(A)) _destr_t ν(T(A))_destr
(ν(T(A)) t){
  ν(Tree(A))_destr_t res;
  res.label = t.label;
  res.kind = t.kind;
  if(res.kind == NODE) {
    res.d1 = (ν(◇))t.left;
    res.d2 = (ν(◇))t.right;
    res.left = *(ν(T(A)) *)t.left;
    res.right = *(ν(T(A)) *)t.right;
  }   return res; }
```

We define $[\![\text{types}(P)]\!]^{\mathtt{C}}$ as the declaration

```
typedef enum {NIL, CONS, LEAF, NODE} kind_t;
```

followed by the declarations $[\![A]\!]^{\mathtt{C}}$ for $A \in \text{types}(P)$ ordered in such a way that $[\![\text{types}(P)]\!]^{\mathtt{C}}$ is valid `C` code (this means that if type A occurs as a subexpression in type B then $[\![A]\!]^{\mathtt{C}}$ must occur before $[\![B]\!]^{\mathtt{C}}$.)

To each declaration $f : (A_1, \dots, A_n) \rightarrow B$ in Σ we associate a prototype $[\![f]\!]^{\mathtt{C}}$ given by $\nu(B)$ f`(`$\nu(A_1)$ x_1`, ... ,` $\nu(A_n)$ x_n`)`

The interpretation of the signature $[\![\Sigma]\!]^{\mathtt{C}}$ is given as the concatenation of $[\![f]\!]^{\mathtt{C}}$; (note the semicolon) for $f \in \mathrm{dom}(\Sigma)$.

A `C`-valuation of a context Γ is a finite function mapping identifiers in $\mathrm{dom}(\Gamma)$ to syntactically correct `C`-expressions.

To each expression $\Gamma \vdash_\Sigma e : B$ and `C`-valuation η satisfying Γ we associate a `C` expression $[\![e]\!]^{\mathtt{C}}_\eta$ of type $\nu(B)$. This definition will rely on the presence of certain `C` variables needed to store intermediate results. When we write something like "where $\nu(A)\, p$ fresh" we mean that p should be a variable of type $\nu(A)$ which is not used anywhere else in $[\![e]\!]^{\mathtt{C}}$. This could be formalised by defining $[\![e]\!]^{\mathtt{C}}$ as a pair $\langle V, D \rangle$ where V maps `C`-valuations to `C` expressions and where D maps sequences of variable declarations to sequences of variable declarations in such a way that $D(s)$ always contains s. We refrain, however, from such a formalisation to reduce notational clutter. We also remark that the only reason why we need these auxiliary variables is that `C` does not provide a construct allowing one to access several components of a structure while evaluating it only once.

The definition of $[\![e]\!]^{\mathtt{C}}_\eta$ is by induction on typed expressions. Recall the following usage of the comma operator in `C` which allows one to simulate "let": an expression of the form $(x{=}\mathit{exp}_1, \mathit{exp}_2)$ is evaluated by first assigning the value of exp_1 to the variable x and then evaluating exp_2 (which may involve x). Unlike in the case of "let" the variable x must have been declared beforehand.

$$[\![x : A]\!]^{\mathtt{C}}_\eta = \eta(x)$$

$$[\![f(e_1, \dots, e_n) : A]\!]^{\mathtt{C}}_\eta = f\texttt{(}[\![e_1]\!]^{\mathtt{C}}_\eta\texttt{, ... , }[\![e_n]\!]^{\mathtt{C}}_\eta\texttt{)}$$

$$[\![c : \mathsf{N}]\!]^{\mathtt{C}}_\eta = c$$

$$[\![e_1 \star e_2 : \mathsf{N}]\!]^{\mathtt{C}}_\eta = \texttt{(}[\![e_1]\!]^{\mathtt{C}}_\eta \star [\![e_2]\!]^{\mathtt{C}}_\eta\texttt{)}$$

$$[\![\mathsf{if}\ e\ \mathsf{then}\ e'\ \mathsf{else}\ e'' : A]\!]^{\mathtt{C}}_\eta = \texttt{(}[\![e]\!]^{\mathtt{C}}_\eta\ \texttt{?}\ [\![e']\!]^{\mathtt{C}}_\eta\ \texttt{:}\ [\![e'']\!]^{\mathtt{C}}_\eta\texttt{)}$$

$$[\![e \otimes e' : A \otimes B]\!]^{\mathtt{C}}_\eta = \nu(A \otimes B)\texttt{\_pair(}[\![e]\!]^{\mathtt{C}}_\eta\texttt{, }[\![e']\!]^{\mathtt{C}}_\eta\texttt{)}$$

$$[\![\mathsf{match}\ e\ \mathsf{with}\ x \otimes y \Rightarrow e']\!]^{\mathtt{C}}_\eta = \texttt{(}p\texttt{=}[\![e]\!]_\eta\texttt{,}[\![e']\!]_{\eta'}\texttt{)}$$
where $\eta' = \eta[x \mapsto p\texttt{.fst}][y \mapsto p\texttt{.snd}]$
where $e : P$ and $\nu(P)\, p$ is fresh.

$$[\![\mathsf{inl}_{A,B}(e) : A+B]\!]^{\mathsf{C}}_{\eta}\nu(A+B)\texttt{\_inl(}[\![e]\!]_{\eta}\texttt{)}$$

$$[\![\mathsf{inr}_{A,B}(e) : A+B]\!]^{\mathsf{C}}_{\eta}\nu(A+B)\texttt{\_inr(}[\![e]\!]_{\eta}\texttt{)}$$

$$\begin{array}{l}[\![\mathsf{match}\ e\ \mathsf{with}\ \mathsf{inl}(x) \Rightarrow e' | \mathsf{inr}(x) \Rightarrow e'' : C]\!]^{\mathsf{C}} = \\ \quad \texttt{(}[\![e]\!]_{\eta}\texttt{.kind==INL ?} \\ \qquad [\![e']\!]_{\eta[x \mapsto [\![e]\!]_{\eta}\texttt{.body.inl}]} \ \texttt{:} \\ \qquad [\![e'']\!]_{\eta[x \mapsto [\![e]\!]_{\eta}\texttt{.body.inr}]}\texttt{)}\end{array}$$

$$[\![\mathsf{nil}_A : \mathsf{L}(A)]\!]^{\mathsf{C}}_{\eta} = \nu(\mathsf{L}(A))\texttt{\_nil()}$$

$$[\![\mathsf{cons}(e_d, e_h, e_t)]\!]^{\mathsf{C}}_{\eta} = \nu(\mathsf{L}(A))\texttt{\_cons(}[\![e_d]\!]^{\mathsf{C}}_{\eta}\texttt{,}[\![e_h]\!]^{\mathsf{C}}_{\eta}\texttt{,}[\![e_t]\!]^{\mathsf{C}}_{\eta}\texttt{)}$$

$$\begin{array}{l}[\![\mathsf{match}\ e\ \mathsf{with}\ \mathsf{nil} \Rightarrow e_{\mathsf{nil}} | \mathsf{cons}(d,h,t) \Rightarrow e_{\mathsf{cons}} : B]\!]^{\mathsf{C}}_{\eta} = \\ \quad \texttt{(}p\texttt{=}\nu(\mathsf{L}(A))\texttt{\_destr(}[\![e]\!]^{\mathsf{C}}_{\eta}\texttt{),}p\texttt{.kind==NIL ? } [\![e_{\mathsf{nil}}]\!]^{\mathsf{C}}_{\eta} \ \texttt{:} \ [\![e_{\mathsf{cons}}]\!]^{\mathsf{C}}_{\eta'}\texttt{)}\end{array}$$

where $\eta' = \eta[d \mapsto p.\texttt{d}, h \mapsto p.\texttt{hd}, t \mapsto p.\texttt{tl}]$
and $\nu(\mathsf{L}(A))\texttt{\_destr\_t}\ \ p$ is fresh

$$[\![\mathsf{leaf}(e) : \mathsf{T}(A)]\!]^{\mathsf{C}}_{\eta} = \nu(\mathsf{T}(A))\texttt{\_leaf(}[\![e]\!]^{\mathsf{C}}_{\eta}\texttt{)}$$

$$\begin{array}{l}[\![\mathsf{node}(e_{d1}, e_{d2}, e_a, e_l, e_r)]\!]^{\mathsf{C}}_{\eta} = \\ \quad \nu(\mathsf{T}(A))\texttt{\_node(}[\![e_{d1}]\!]^{\mathsf{C}}_{\eta}\texttt{,} \ [\![e_{d2}]\!]^{\mathsf{C}}_{\eta}\texttt{,}[\![e_a]\!]^{\mathsf{C}}_{\eta}\texttt{,}[\![e_l]\!]^{\mathsf{C}}_{\eta}\texttt{,}[\![e_r]\!]^{\mathsf{C}}_{\eta}\texttt{)}\end{array}$$

$$\begin{array}{l}[\![\mathsf{match}\ e\ \mathsf{with}\ \mathsf{leaf}(a) \Rightarrow e_{\mathsf{leaf}} | \mathsf{node}(d_1, d_2, a, l, r) \Rightarrow e_{\mathsf{node}} : B]\!]^{\mathsf{C}}_{\eta} = \\ \quad \texttt{(}p\texttt{=}\nu(\mathsf{L}(A))\texttt{\_destr(}[\![e]\!]^{\mathsf{C}}_{\eta}\texttt{),}p\texttt{.kind==LEAF ? } [\![e_{\mathsf{leaf}}]\!]^{\mathsf{C}}_{\eta[a \mapsto p.\texttt{label}]} \ \texttt{:} \\ \qquad [\![e_{\mathsf{node}}]\!]^{\mathsf{C}}_{\eta[d_1 \mapsto p.\texttt{d1}, d_2 \mapsto p.\texttt{d2}, a \mapsto p.\texttt{label}, l \mapsto p.\texttt{left}, r \mapsto p.\texttt{right}]}\end{array}$$

where $\nu(\mathsf{T}(A))\texttt{_destr_t}\,p$is fresh

The interpretation $[\![P]\!]^{\mathsf{C}}$ of the *program* $P = (\Sigma, (e_f)_{f \in \mathrm{dom}(\Sigma)})$, finally, is obtained as the concatenation of the following blocks:

1. the interpretation of the types: $[\![\mathrm{types}(P)]\!]^{\mathsf{C}}$,
2. the function prototypes: $[\![\Sigma]\!]^{\mathsf{C}}$,
3. for each $f \in \mathrm{dom}(\Sigma)$ a function definition

   $[\![f]\!]^{\mathsf{C}}$ `{`
   $\quad DECL(e)$
   $\quad$ `return` $[\![e]\!]^{\mathsf{C}}_{[x_1 \mapsto x_1, \ldots x_1 \mapsto x_1]}$`;`
   $\}$

   where $DECL(e)$ consists of all the variable declarations which have occurred in the definition of $[\![e]\!]^{\mathsf{C}}$.

2.3 Examples

Although the language presented above is comparatively simple and is still experimental, its power of expression is still impressive, as many non trivial functions may be formulated with it. Common examples of such functions are exhibited below.

Reverse:

$$
\begin{array}{l}
\texttt{rev\_aux} : (\mathsf{L}(\mathsf{N}), \mathsf{L}(\mathsf{N})) \rightarrow \mathsf{L}(\mathsf{N}) \\
\texttt{reverse} : (\mathsf{L}(\mathsf{N})) \rightarrow \mathsf{L}(\mathsf{N}) \\
e_{\texttt{rev\_aux}}(l, acc) = \mathsf{match}\ l\ \mathsf{with} \\
\quad\quad \mathsf{nil} \Rightarrow acc \\
\quad\quad |\mathsf{cons}(d, h, t) \Rightarrow \texttt{rev\_aux}(t, \mathsf{cons}(d, h, acc)) \\
e_{\texttt{reverse}}(l) = \texttt{rev\_aux}(l, \mathsf{nil}_{\mathsf{N}})
\end{array}
$$

Quick sort:

$$
\begin{array}{l}
\texttt{append} : (\mathsf{L}(\mathsf{N}), \mathsf{L}(\mathsf{N})) \rightarrow \mathsf{L}(\mathsf{N}) \\
\texttt{split} : (\mathsf{N}, \mathsf{L}(\mathsf{N})) \rightarrow \mathsf{L}(\mathsf{N}) \otimes \mathsf{L}(\mathsf{N}) \\
\texttt{sort} : (\mathsf{L}(\mathsf{N})) \rightarrow \mathsf{L}(\mathsf{N}) \\
e_{\texttt{append}}(l, r) = \mathsf{match}\ l\ \mathsf{with} \\
\quad\quad \mathsf{nil} \Rightarrow r \\
\quad\quad |\mathsf{cons}(d, h, t) \Rightarrow \mathsf{cons}(d, h, \texttt{append}(t, r)) \\
e_{\texttt{split}}(p, l) = \mathsf{match}\ l\ \mathsf{with} \\
\quad\quad \mathsf{nil} \Rightarrow \mathsf{nil} \otimes \mathsf{nil} \\
\quad\quad |\mathsf{cons}(d, h, t) \Rightarrow \mathsf{match}\ \texttt{split}(p, t)\ \mathsf{with} \\
\quad\quad u \otimes v \Rightarrow \mathsf{if}\ h \leq p \\
\quad\quad\quad \mathsf{then}\ \mathsf{cons}(d, h, u) \otimes v \\
\quad\quad\quad \mathsf{else}\ u \otimes \mathsf{cons}(d, h, v) \\
e_{\texttt{sort}}(l) = \mathsf{match}\ l\ \mathsf{with} \\
\quad\quad \mathsf{nil} \Rightarrow \mathsf{nil} \\
\quad\quad |\mathsf{cons}(d, h, t) \Rightarrow \mathsf{match}\ \texttt{split}(h, t)\ \mathsf{with} \\
\quad\quad u \otimes v \Rightarrow \texttt{append}(\texttt{sort}(u), \mathsf{cons}(d, h, \texttt{sort}(v)))
\end{array}
$$

Breadth-first search:

$$
\begin{array}{l}
\texttt{snoc} : (\diamond, \mathsf{L}(\mathsf{T}(\mathsf{N})), \mathsf{T}(\mathsf{N})) \to \mathsf{L}(\mathsf{T}(\mathsf{N})) \\
\texttt{breadth} : (\mathsf{L}(\mathsf{T}(\mathsf{N}))) \to \mathsf{L}(\mathsf{N}) \\
e_{\texttt{snoc}}(d, l, t) = \mathsf{match}\ l\ \mathsf{with} \\
\qquad \mathsf{nil} \Rightarrow \mathsf{cons}(d, t, \mathsf{nil}()) \\
\qquad |\mathsf{cons}(d', t', q) \Rightarrow \mathsf{cons}(d', t', \texttt{snoc}(d, q, t)) \\
e_{\texttt{breadth}}(q) = \mathsf{match}\ q\ \mathsf{with} \\
\qquad \mathsf{nil} \Rightarrow \mathsf{nil} \\
\qquad |\mathsf{cons}(d, t, q) \Rightarrow \mathsf{match}\ t\ \mathsf{with} \\
\qquad \quad \mathsf{leaf}(a) \Rightarrow \mathsf{cons}(d, a, \texttt{breadth}(q)) \\
\qquad \quad |\mathsf{node}(d_1, d_2, a, l, r) \Rightarrow \mathsf{cons}(d, a, \\
\qquad \qquad \texttt{breadth}(\texttt{snoc}(d_2, \texttt{snoc}(d_1, q, l), r)))
\end{array}
$$

3. Design

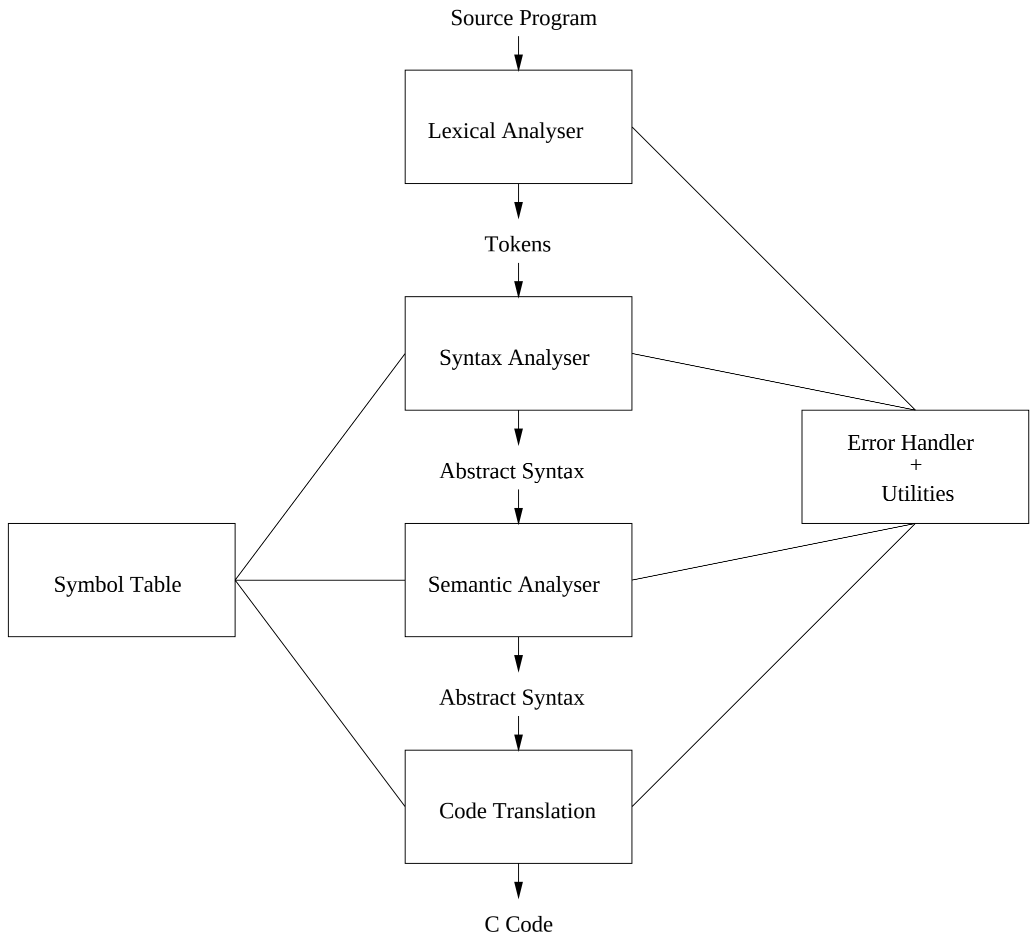
Although the definition of the language as presented in the last chapter is rigorous in it's treatment of the typing rules and how the translation should be carried out, it does not cover all the details require to make a complete implementation. It's treatment is mathematical in nature and does not specify such details as structure of a program or concrete syntax for the language. Programs are considered as a signature followed by definitions of each of the function symbols, without concern for how they should be formatted. Similarly the exact layout of functions is not given, nor how type are to be represented. Such details are discussed below and the extra specifications need to build a complete system are given.

3.1 Structure

The design of the compiler follows the standard practice of breaking the process of compilation into a series of phases. Each of the phases represents a logically distinct stage of the transformation of source code into target code. This follows closely the structure for compiler construction outlined in [1] and [2]. Breaking the compiler into stages allows making modifications easier as only the effected stage need be altered. This is particularly the case with experimental systems, where changes are made regularly. It also allows new feature to be added without having to modify the whole compiler.

The process can be clearly separated into four distinct phases that represent different levels of the transformation of source code into target code. The first three stages, the front end, are concerned with analysis of valid source programs, and the last, the back end, is concerned with synthesis of target code. The analysis phases are characterised by breaking up of the source code into parts, the construction of an intermediate representation and the maintenance of information about identifiers. The synthesis stage constructs the target code from the intermediate representation.

The structure of the compiler, showing each of the phases, can be seen in the diagram below. Although the symbol table and error handler are not strictly phases in the compilation process they are shown here to indicate how they interact with the various parts of compiler.



Structure of the Compiler

3.1.1 Lexical Analysis

The lexical analyser will accept as input a stream of characters representing the source code and produce a stream of lexical tokens as output. Some of these tokens will have lexical values associated with them. The Tokens which are recognised by the language are detailed below. They correspond to keywords and punctuation within the language outlined in the previous chapter, their relationship, being fairly apparent.

The identifiers are sequences of letters, digits and underscore starting with a letter. Within the context of an identifier, uppercase letters are distinguishable from those in lowercase. There is no limit to length of an identifier. This natural intuition means that the identifiers can be describe by the regular expression;

$$[a - zA - Z][a - zA - Z0 - 9\_]*$$

The reserved words of the language are;
`fun, list, tree, int, diamond, match, with, if, then, else, nil, cons, leaf, node`.
These reserve word may not be used as identifiers, as it would make language harder for a programmer to understand and may lead to ambiguities. Although the programmer is restricted in chose of identifiers, this is made up by increased clarity and simplifies the complexity of the compiler as otherwise no identifier could be matched without regard to its context if the reserve words are admitted.

The only reserve word that warrants mention is `diamond`. It is used to represent the resource type $\diamond$, as this is how the symbol is commonly known. The choice of keyword for this type was arbitrary, and any word or symbol could be used. Other choice could be `chunk` or `space`, which would be preferable is best gauged by programmer response.

The punctuation symbols used in the language are;
`:  [ ] ( ) | => = + - * / > < ==`

A number is a sequence of decimal digits and represents the corresponding integer value. The is no negative numbers as again this would increase the complexity of compiler as it would be difficult to distinguish between a negative number and the subtraction operator, thought this could be done by the parser, see section 3.1.2. The regular expression representing the numbers is;

$$[0-9]*$$

To increase the clarity of programs, white space, including newlines and tabs are allowed between any two tokens. This lack of restriction on the layout of programs means that programmers are free to structure the code in any way they choose, so increasing (or decreasing) clarity. Similarly to aid clarity, comments can be placed between any two tokens, each comment starting with `/*` and ending with `*/`. Anything between these tokens is ignored by the compiler. To keep the lexical analysis simple it was decided that nested comments would not be allowed, despite their usefulness to programmers, as admitting them would require keeping a global record of nesting level, or complicate the grammar for the language unnecessarily.

To help with error handling , the lexical analyser will associated with each token it position within the source file. see section 3.2 for details on error handling capabilities.

3.1.2 Syntax Analysis

The syntax of the language of the compiler can be gleamed from the typing rules presented in section 2.1, but there a few details missing to completely specify it. The BNF style Grammar that describes a concrete syntax for the language is given below.

Concrete Syntax [1]

$$
\begin{array}{ll}
Program & ::= \{Signature \mid Definition\}_{0+} \\
Signature & ::= \mathsf{idenifier} : (\{Type\_List\}_{opt})Type \\
Type\_List & ::= Type\{, Type\}_{0+} \\
Type & ::= \mathsf{int} \mid \mathsf{diamond} \mid \mathsf{list}(Type) \mid \mathsf{tree}(Type) \mid [Type, Type] \\
Definition & ::= \mathsf{fun\ identifier}(\{Parameter\_List\}_{opt}) = Expression \\
Parameter\_List & ::= Variable\{, Variable\}_{0+} \\
Variable & ::= \mathsf{identifier} \\
Expression & ::= Variable \\
 & \mid \mathsf{number} \\
 & \mid (Expression) \\
 & \mid Expression\ Binary\_Operator\ Expression \\
 & \mid \mathsf{if}\ Expression\ \mathsf{then}\ Expression\ \mathsf{else}\ Expression \\
 & \mid (Expression, Expression) \\
 & \mid \mathsf{nil}(Type) \\
 & \mid \mathsf{cons}(Epression, Expression, Expression) \\
 & \mid \mathsf{leaf}(Expression) \\
 & \mid \mathsf{node}(Expression, Expression, Expression, Expression, Expression) \\
 & \mid \mathsf{match}\ Expression\ \mathsf{with\ nil}\ => Expression \\
 & \quad \mid \mathsf{cons}(Variable, Variable, Variable) => Expression \\
 & \mid \mathsf{match}\ Expression\ \mathsf{with\ leaf}\ (Variable) => Expression \\
 & \quad \mid \mathsf{node}(Variable, Variable, Variable, Variable, Variable) => Expression \\
 & \mid \mathsf{match}\ Expression\ \mathsf{with}\ (Variable, Variable) => Expression \\
 & \mid \mathsf{identifier}(\{Expression\_List\}_{opt}) \\
Expression\_List & ::= Expression\{, Expression\}_{0+} \\
Binary\_Operator & ::= + \mid - \mid * \mid / \mid == \mid > \mid <
\end{array}
$$

It should be noted that both identifier and number are treated as terminals of the grammar. They are dealt with by the lexical analysis stage and there meaning is given in section 3.1.1. They are the only terminals that have values associated with them.

The range of binary operators chosen was dictated by those available in C. Those chosen are powerful enough for most program to be expressed, although more could be added. There meaning is the same as those in C. As was mentioned earlier there is no negative numbers. It was also decided to omit the unary negation operator from the language as it would over complicate the syntax analysis. However, it is possible to get a similar effect by subtracting a number from zero.

[1] note that out of line |'s are part of the language, not production rules

To aid with ease of programming, signatures and definitions may be freely mixed, with the restriction that the definition of a particular function is given after its signature. However this condition does not effect the parsing of the language.

The type N is represented by int as is commonly used in other languages. Pair types, $\otimes$ are presented by bracketing comma separated subtypes. This is not the syntax familiar to most, but it is intuitive and simplifies syntax analysis as using the normal * syntax may lead to ambiguities with the symbol for multiplication.

The syntax for the expressions matches the syntax used in the typing rules and should be recognisable to most. The only difference to the typing rules is in the case of nil, which now must be annotated with a type. It was decided that this should be done, although inconvenient for the programmer as it will greatly simplify the type checking of expression. Any expression may also be enclosed with braces () to help make the meaning of a program clear.

Although this grammar completely describes the syntax of the language it is not without its ambiguities. The rules for expressions have shift/reduce errors. These are caused by the match and if constructs and the binary operators. These are well understood ambiguities, so are acceptable. An explanation of how they are dealt with is given in section 4.3.

Despite the fact the above concrete syntax grammar has ambiguities, the parse tree that it describes is inconvenient for later stages of the compilation process to use as it contains redundant punctuation tokens that convey no information once a concrete parse tree is constructed. A more abstract syntax parse tree would separate the issues of parsing from the essential structure of the program. Such a abstract syntax tree structure, that the semantic analysis and translation phase can traverse, simplifies coding.

The abstract grammar that represents our language is give below. This grammar is completely impractical to use for parsing as it is very ambiguous, since precedence of productions it not given and many punctuation and keyword tokens are missing. However it is not meant for parsing. The parser will use the concrete syntax given above to a create a parse tree for the abstract syntax. This abstract syntax tree will be used by later phases of compiler. They are only concerned with the structure of a program, and this is completely described by a parse tree in the abstract syntax.

Abstract Syntax [2]

program	::= *sig_def_list*
sig_def_list	::= *signature_or_definition sig_def_list*
	$\mid \epsilon$
signature_or_definition	::= *signature* | *definition*
signature	::= indentifer *type_list type*
definition	::= indentifer *variable_list expression*
variable_list	::= *variable variable_list*
	$\mid \epsilon$
variable	::= identifer *type*
type_list	::= *type type_list*
	$\mid \epsilon$
type	::= int | diamond | list *type* | tree *type* | pair *type type*
expression_list	::= *expression expression_list*
	$\mid \epsilon$
expression	::= *variable*
	| number
	| *expression operator expression*
	| if *expression expression expression*
	| pair *expression expression*
	| nil *type*
	| cons *expression expression expression*
	| leaf *expression*
	| node *expression expression expression*
	expression expression
	| matchlist *expression expression variable*
	variable variable expression
	| matchtree *expression variable expression*
	variable variable variable variable variable expression
	| matchpair *expression variable expression expression*
	| function identifer *expression_list*
operator	$::= + \mid - \mid * \mid / \mid == \mid > \mid <$

3.1.3 Semantic Analysis and the Symbol Table

This phase of compilation process is characterised by the maintenance of symbol tables mapping identifiers to types and location. As variables and functions are declared these identifiers are bound to their type in the symbol table.

Most of the type checking rules care given in chapter 2 and do require further explanation, however there are few points that warrant further mention, due their uniqueness or effect on the implementation of the compiler.

The basic types for our language are `int` and $\diamond$. The other types are constructed using List , Tree or Pair `[a,b]` from other types.

[2]note that some of lines are continuations

There are two name spaces with in our language, the variables, and the functions, we therefore maintain two symbol tables, one for each. We could have single symbol table, however it makes implementation simpler if all the entries in a symbol are of the same sort.

Variables are bound to their type when they are declared. Similarly functions are bound to their return type and a list of their parameter types when they are declared.

As well as declarations, `match` constructs bind variables. Because `match` constructs may be nested, our symbol table for variables must be able to handle nested scope.

The semantic analyser will recurse over the abstract syntax parse tree constructed by the parser, type checking each of nodes. Each of the productions in the abstract grammar, given earlier, corresponds to one of the typing rules given in chapter 2. The correspondence should be apparent ,with exception of list productions. However these can be type checked by recursively checking its components, and doing nothing for the empty production. The type rule for a definition is given by the description of a program.

3.1.3.1 Disjointness Test

The special feature of the language that make it unique, is that functions and operations uses their arguments and operands at most once, that is, in a linear fashion. This warrants special further attention.

When an expression is type checked it is necessary to check that heap allocated variables (list, tree, and possibly pairs) that are in the scope of the whole expression are only within the scope of at most one of its sub components. This can be achieved by building sets of variables used at each node as the parse tree is traversed. These sets can then be checked for dis-jointness. This requires the construction of a abstract data type (ADT) to represent these sets.

3.1.4 Code Translation

The final stage of the compiler synthesises the C code by traversing the abstract syntax parse tree in a similar fashion to the semantic analyser. It also make use of a symbol table, mapping source code identifiers to there C code representation. The C code is produced as a `string`. The only major point that will effect a design decision is that a lot of string manipulation will be required during the traversal of the parse tree. As strings are static, this will involve much copying and creation of new strings so that they may be joined. It is therefore prudent at this stage to require that a class of functions be implemented to perform these string operations. As many strings are likely to be involved it is also prudent to require such functions to `free()` strings they are finished with. see 3.3

3.1.4.1 Intermediate Results

As mention in section 2.2, the target language, C, does not have any constructs[3] that bind variables in a similar way to the `match` construct in the functional language, the code to be produced will contain a lot of function-global variables to hold intermediate results.These variables must be uniquely identified. This requires the compiler to be able to produce unique, valid C identifiers on demand. It should be noted that identifiers need only be unique within a function, different function may use the same identifier. This can be achieved by using a counter as the C string representation of an identifier, prefixing it with a letter to make it valid. This counter in incremented after every use, and can be reset to 0 for each new function definition.

This only gives a finite number of identifiers, the size of the largest number that be stored, however this should suffice for this compiler, and to simplify implementation, no checking for overflows will be made.

3.2 Error Handling

It's useful (if not essential) for a compiler to handles errors, simply producing no output does not help the programmer. By producing meaningful error messages, the compiler enables a programmer to use the system in an interactive fashion, adjusting and improving code in an iterative fashion, until it is correct. This is particularly true in this case as the programming language given here is new and while it's constructs will be familiar to most, it does include a new paradigm, explicit space allocation, which requires learning. To help the programmer to use the system, meaningful error messages are required. A message should at the least indicate the location and type of error.

In a one pass compiler, where all of tasks mentioned above are carried out simultaneously, then if an error occurs (for example a typing or parse error) that must be reported to the programmer, then the current position of the lexical analyser will indicated the source position of the error.

However in the our compiler, which use an abstract-syntax-tree data structure and carries out the compilation in several stages things are more complicated.

[3] except for functions

The lexical analyser has reach the end of source code before any on the next stages have began, so if an error is detected while traversing the abstract syntax tree then using the current position of lexer, which is the end of file, will produce meaningless error messages. It is therefore necessary to mark each node of the abstract syntax tree with its source code position, in case it contains an error.

Our system will always produce a message, when a error is detected. It will indicated the position it was detected, the phase of compilation it occurred in, and the type of the error. This is relatively basic information, as no details on why a particular error happened are given. However as this compiler is experimental and the project is more concerned with demonstrating that a it can be implemented than the details this will be sufficient.

With similar reasoning it was decided that error recovery would be left out, though it could be easily included. The compiler will not attempt to recover from a parse error by using error productions, or type error by inserting the expected type. When a error is detected the system will report it and then halt. This is acceptable as further error messages are likely to be less useful as they are likely caused by the knock on effect of the first.

3.3 Memory Considerations

The compiler uses complex data structures to manage the translation process. Both the abstract syntax and symbol table involve the use of mutually recursive data structures to represent the parse tree and type environment respectfully.

The use of such structures must done carefully. Constructor functions are used throughout the compiler for there creation. On the main, once the abstract syntax parse tree is constructed it remain static. To this end it was decided that careful deconstruction would be overlooked to ease the writing of the rest of the compiler.

No destructor functions are declared for abstract syntax data types, so deallocating them is not possible. This has little effect on the system as the parse tree is static once created. The exception is when types are created for insertion into the symbol table as in this case they will eventually be removed, and may be deallocated. This was over looked, resulting in a potential space leak. However as we are concern with showing that a compiler could be implemented it was deemed acceptable, to speed along the implementation of a complete system. An industrial strength version of this compiler would of course be more careful with its memory usage.

The exception to this rule will be the strings representing the C code ,mentioned previously, `free`-ing two strings, once they are copied into a larger space is straight forward

3.4 Examples

The language as designed is easily understood, as it allow functions to written in a relatively intuitive way that will be familiar to programmers that have used similar functional languages such as SML and OCAML. The example functions presented in section 2.3 can be seen below, written in our new concrete syntax for the language.

Reverse:

```
rev_aux:(list(int),list(int))list(int)
reverse:(list(int))list(int)

fun rev_aux(l,acc) = match l with
        nil => acc
        | cons(d,h,t) => rev_aux(t,cons(d,h,acc))
fun reverse(l) = rev_aux(l,nil(int))
```

Quick sort:

```
append:(list(int),list(int))list(int)
split:(int,list(int))[list(int),list(int)]
sort:(list(int))list(int)

fun append(l,r) = match l with
        nil => r
        | cons(d,h,t) => cons(d,h,append(t,r))

fun split(p,l) = match l with
        nil => [nil(int) ,nil(int)]
        | cons(d,h,t) => match split(p,t) with
                        [u,v] => if h<p then [cons(d,h,u),v]
                                        else [u,cons(d,h,v)]
fun sort(l) = match l with
        nil => nil(int)
        | cons(d,p,t) => match split(p,t) with
                        [u,v] => append(sort(u),cons(d,p,sort(v)))
```

Breadth-first search

```
breadth:(list(tree(int)))list(int)

snoc:(diamond,list(tree(int)),tree(int))list(tree(int))

fun snoc(d,l,t) = match l with
        nil => cons(d,t,nil(tree(int)))
        |cons(d1,t1,q) => cons(d1,t1,snoc(d,q,t))

/* this is a comment*/

fun breadth(q) = match q with
        nil => nil(int)
        |cons(d,t,q) => match t with
                leaf(a) => cons(d,a,breadth(q))
                |node(d1,d2,a,l,r) => cons(d,a,
                  breadth(snoc(d2,snoc(d1,q,l),r)))
```

4. Implementation

C was chosen as the implementation language as it the language I am the most familiar with, and in implementing a large system I felt it best to stick with what I can use fell. Using C does have it advantages. It is a very flexible system, so all constructs need to build the system could be synthesised. Many other compiler systems have been built using C and there a many tools available to to help with the construction of the various phases of of a compiler.

A few points on the style of coding used are worth mentioning. Each phase of the compiler and each of the abstract data types used, is located in it's own C file with a separate header file describing the functionality that it provides . All functions and global variables made available my a module through its header, share a common prefix to their names. This means that it can be determined where a name comes from without having to examine multiple files.

All data structures are provided with constructor function, so that internal representation can be ignored. Further all functions have prototypes declared for them so that may be used freely within the code without concern about the order in which they are declared.

The dependencies between the various modules of the system can be seen in the Makefile in appendix A.27.

4.1 Error Handler and Utility Functions

As was shown in the diagram of the compiler structure in the previous chapter the error handler and a few utility functions are used by all phases of the compilation process. The error handling code allow each stage to report complication error in a consistent manner. The utility functions simplify coding and deal with memory issues.

The error handling code was taken from [2], and a complete description of its working can be found there. The mains points that are relevant to our system are that the lexical analyser passes the position of each token ,represented as a character position within the source, to the parser which uses this when constructing the abstract syntax parse tree, and that a global list is kept of the length of each line of source code. On matching a token the lexical analysis stage calls the function,

```
void adjust{void};
```

which increments the global variable

```
int EM_tokPos;
```

by the length of the token. The parser has access to this variable and uses it to associated a position with each of the nodes of the abstract syntax parse tree as it constructs them. When the lexer encounters a newline it calls the function,

```
void EM_newline(void);
```

this stores the current position into a global list,

```
typedef struct intList {int i; struct intList *rest;} *IntList;
static IntList linePos=NULL;
```

and increments the global count of lines,

```
static int lineNum = 1;
```

Using this list it is possible to determine for a given source code character position it's line, and position within that line. This list is used by the error reporting function,

```
void EM_error(int pos, char *message,...)
```

When an error occurs in one of phases, this function is called with the position of the error error, which it gets from the position element of the appropriate node of abstract syntax parse tree, and a meaningful error message. This message can be formatted in a similar way to output of a `printf` call. The error function can then print this message along with line position information. It then terminates the compilation process.

The utilities module defines;

```
typedef char *string;

string String(char *s) {
        string p = checked_malloc(stlen(s)+1);
        strcpy(p,s);
        return p;
}
```

This type definition is included because the C code that is constructed by the compiler is represented by a string and this saves confusion using pointers to characters. However, as strings are heap allocated, functions that take these as arguments must assume that they are static. However during the construction of the C code the strings representing fragments must be joined and grow. For details on how this is dealt with see section 4.6.

Another function that is included as a general utility helps with memory management. C's `malloc()` function will return `NULL` if there is no memory left. As mentioned earlier, our compiler does not deallocated certain data structures when it is finished with them. This lack of sophisticated memory management could result in the system running out of memory. To deal this potential problem the compiler never calls `malloc` directly, but instead the function `checked_malloc`, which will never return `NULL`,

```
void *checked_malloc(int len) {
        void *p = malloc(len);
        assert(p);
        return p;
}
```

4.2 Lexical Analysis

Flex, the free-ware implementation of lex was used to generate a lexical analyser for the compiler. For details on Flex and its use see [9]. It automatically produces a C program that simulates a Deterministic Finite Automata (DFA) that matches tokens. The tokens to be matched are specified as regular expression. Connected to each regular expression is a C code statement that returns the token matched.

Part of the statement for each regular expression is the `adjust()` function discussed above. As an example of the format of a line in flex specification file consider the line that matches identifiers,

```
[a-zA-Z][a-zA-Z0-9_]* {adjust();yylval.sval=String(yytext);return ID;}
```

As mention earlier, as well as identifiers, numbers are the only tokens that have a value associated with them. The identifiers are stored are stored a `string` type and the numbers as `int`. As the lexer matches a stream of characters, which is stored in `yytext`, it is necessary to convert this to the appropriate type. Above, the utility function `String()` is used and for numbers a call to `atoi()`.

The token that is returned is actually a `union` of types, `yylval`. The nature of this structure is discussed in the next section. The value connected with a token must be placed into this union.

The special command `continue` within a action causes the lexer to consumes its input without return a token type. This is used in our compiler to ignore 'white space', and comments.

```
" " {adjust();continue;}
"\n" {adjust(); EM_newline(); continue;}
```

```
"\t" {adjust(); continue;}
```

Note the call to `EM_newline()` mentioned in the last section.

The matching of comments warrants particular mention. As a comment may contain any of the tokens, it is necessary to separate them from the rest of the regular expressions. This is achieved by the use of 'start conditions'. The comment state is entered when `/*` is matched and input consumed until `*/` is matched when it is left. When within a state only those regular expressions that are prefixed by it are considered.

```
"/*" {adjust(); BEGIN COMMENT;}
<COMMENT>"*/" {adjust(), BEGIN(0);}
<COMMENT>. {adjust(); continue;}
```

The order which regular expressions are listed is important, if two expressions match the input, then first one is accepted. This means that if all the keywords are listed before the expression that matches identifiers then these keywords will not be accepted as identifiers. This is part of our specification as mention in the previous chapter.

The regular expression that matches any character is placed at the end, where it its action will be executed if no other expression matches, which is the case for invalid input. It calls `EM_error()` to indicate this.

```
.  {adjust(); EM_error(EM_tokPos, "what kind of token is that!");}
```

The complete input file for Flex, can be seen in Appendix A.6.

4.3 Parsing

To implement the parser discussed in section 3.1.2 Bison the GNU parser generator was used. For details on Bison see [3]. The grammar as given cannot be used directly by Bison as a few of the descriptive constructs used do not have representations within Bisons description language. A few alterations must be made before it will be accepted.

The grammar given previously also contain a few ambiguities that would result in shift/reduce errors within the parser. These can be removed by specifying the precedence and associativity of tokens.

```
%right MATCH WITH GOESTO BAR
%right IF THEN ELSE
%left CONS NODE LEAF
%left LESS_THAN GREATER_THAN SAME_AS
%left PLUS MINUS
%left MULTIPLY DIVIDE
```

Those lower down the list have a higher precedence. The rules given here follow the natural intuition about the associativity and precedence of operators. The binary operators associate to the left and multiplication and division bind more tightly that addition and subtraction, and so on. In the case of `if` and `match` expressions, we follow the normal practice of associating an else or with token (etc) with the if statement closest to it on the left. All these rules are sufficient to remove all ambiguities from the parser.

All the tokens and non-terminals have a semantic value and each each of the productions in the grammar has a semantic action associated with it. The semantic value of each, in the case of our parser, is one of abstract syntax types mention in the next section. When a production matches, it's semantic action is executed. In this way the abstract syntax tree can be built,

```
definition: FUN ID LEFT_PAR parameter_list RIGHT_PAR EQUAL expression
  {$$=A_Definition(EM_tokPos,S_Symbol($2),$4,$7);}
```

The complete input file for Bison can be seen in Appendix A.5

4.3.1 Abstract Syntax

The abstract syntax is represented by mutually recursive data types created as structures. It is best described by giving an example and listing each of the structures. There are twelve data types, one for each of the non-terminals given in the abstract syntax.

```
typedef struct A_program_ *A_program;
typedef struct A_signature_definition_ *A_signature_definition;
typedef struct A_signature_ *A_signature;
typedef struct A_definition_ *A_definition;
typedef struct A_variable_ *A_variable;
typedef struct A_type_ *A_type;
typedef struct A_expression_ *A_expression;

typedef struct A_sig_def_list_ *A_sig_def_list;
typedef struct A_variable_list_ *A_variable_list;
typedef struct A_type_list_ *A_type_list;
typedef struct A_expression_list_ *A_expression_list;

typedef enum {A_plusOp, A_minusOp, A_multiplyOp, A_divideOp, A_lessOp,
A_greaterOp, A_sameOp} A_oper;
```

As an example of one these structures consider,

```
struct A_definition_
{
  A_pos pos;
  S_symbol name;
  A_variable_list parameter_list;
```

```
  A_expression body;
};
```

As can be seen, this data structure mimics that of its counter part production in the abstract syntax given, augmented by its position. In the case of a non-terminal that has several productions, each is represented by a member of a `union` and which type in particular is represented is given by a enumerated kind field.

Constructor functions are provided for each of the productions in the abstract grammar,

```
A_definition A_Definition(A_pos pos, S_symbol name, A_variable_list parms,
A_expression body);
```

These functions are called within the semantic actions of the parser to built the abstract syntax parse tree.

The structure associated each non-terminal can be seen in appendix A.1.

4.3.2 Pretty Printer

To help with debugging, both by the user of the compiler and future implementors, functions which show the structure and contents of an abstract syntax tree produced by the compiler were created.

```
void pretty_print_program(FILE *out, A_program v, int indentation);

void pretty_print_expression(FILE *out, A_expression v, int indentation);
```

These functions recurse over the syntax tree, given by `v`, and produce a nicely formatted representation of this in the location specified by `out`. This can of course be standard input or error. `indentation` is used internally to produce the correct level of indentation for a line of output, when called externally it should be passed `0`. It is only visible because of the recursive nature of the functions, they could be hidden behind dummy functions headers.

These functions may be useful for future development of the system. They can currently be accessed as flag option within the compiler, see the introduction at the start of Appendices for details. Below is the formatted representation of the abstract syntax tree produced after parsing the `reverse()` function given previously.

```
 Program(
  Sig and Def List(
   Sig(
    Signature(rev_aux,
     Type List(
      List(
       Int),
```

```
      Type List(
       List(
        Int),
             )
     )
     List(
      Int))
)
,
   Sig and Def List(
    Sig(
     Signature(reverse,
      Type List(
       List(
        Int),
             )
      List(
       Int))
)
,
    Sig and Def List(
     Def(
      Definition(rev_aux,
       Variable List(
        Variable(l),
        Variable List(
         Variable(acc),
                 )
       )
       Match_ListExp(
        VariableExp( Variable(l)),
        VariableExp( Variable(acc)),
        Variable(d),
        Variable(h),
        Variable(t),
        Function_CallExp(rev_aux
         Expression List(
          VariableExp( Variable(t)),
          Expression List(
           ConsExp(
            VariableExp( Variable(d)),
            VariableExp( Variable(h)),
            VariableExp( Variable(acc))),
                     )
         )
        )))
)
,
     Sig and Def List(
      Def(
       Definition(reverse,
```

```
        Variable List(
         Variable(l),
                 )
        Function_CallExp(rev_aux
         Expression List(
          VariableExp( Variable(l)),
          Expression List(
           NilExp(
            Int),
                     )
         )
        ))
)
,
      )
)
)
)
)
```

4.4 The Symbol Table

The implementation of the symbol tables used through out this compiler is taken from [2]. A complete explanation of it's working can found in chapter 5 of that book.

The key points are that it is implemented as a hash table in which `S_symbol`'s can be mapped to anything. This allows us to create multiple symbol tables each for a different purpose without having to have different implementations. Instead of using string identifiers it uses.

```
typedef struct S_symbol_ {string name; S_symbol next;} *S_symbol;
```

These can be hashed into the table more efficiently, and allows for fast comparisons. The functions,

```
S_symbol S_Symbol(string);
string S_name(S_symbol);
```

will allow the conversion of a string identifier to and from one of these symbols. The symbol table,

```
typedef struct TAB_table_ *S_table;
```

admits the following functions which provide all the functionality discussed in section 3.1.3.

```
S_table S_empty(void);
void S_enter(S_table t, S_symbol sym, void *value);
void *S_look(S_table t, S_symbol sym);
void S_beginScope(S_table t);
```

```
void S_endScope(S_table t);
```

4.5 Semantic Analysis

During this phases we recursively traverse the abstract syntax tree produced by the previous phase. As we do this we maintain two symbol tables, one for the variables and one for function declarations. For each of the abstract data types a checking function is declared, for example,

```
SEM_expression_return SEM_check_expression(SEM_environment environment,
A_expression e);
```

When we are checking a node of the parse tree, we do so within a context. This context is the environment of the node and is presented by our two symbol tables,

```
struct SEM_environment_
{
  S_table variables;
  S_table functions;
};
```

The binding for an identifer within a symbol table is its type, and this is represented by the recursive data type declared for the abstract syntax tree given earlier. Therefore, we must provide a function for comparing two of these data types, so that checks may be performed. We declare the following functions, which recurses over `A_type`

```
int Ty_are_equal(A_type a, A_type b);
int Ty_is_heap(A_type a);
```

In the case of the function symbol table, we bind the function identifier (represented as a symbol) to the following structure,

```
struct Ty_function_
{
  A_type_list formals;
  A_type result;
};
```

With all these definitions in place, we are in position to perform all type checking rules give in section 2.1. See appendix A.12 for the details of the code that performs all the checks.

If the node of the parse tree, and all its sub nodes, are successfully type checked then the function returns with the resulting type of the node,

```
struct SEM_expression_return_
{
  SET_set heap_variables;
  A_type type;
};
```

The use of the heap_variables component of this returned structure is discussed in the next section.

4.5.1 Disjointness Test

The all important feature of the language presented in this paper is that heap allocated variables are used at most once within an expression. This means that a variable should not appear in the set of used variables for two sub expressions.

To this end we maintain sets of those heap allocated variables used by a node as we type check it. This set of heap allocated variables is represented by the ADT given here.

```
typedef struct SET_set_ *SET_set;

struct SET_set_
{
  S_symbol data;
  SET_set next;
};
```

The internal representation of a set is simply a list. Our ADT admits the following operations, each of which should be obvious,

```
SET_set SET_Empty(void);
SET_set SET_Singleton(S_symbol element);
BOOL SET_IsMember(SET_set set, S_symbol element);
SET_set SET_Add(SET_set set, S_symbol element);
SET_set SET_Remove(SET_set set, S_symbol element);
```

The important detail is that SET_ADD should check membership before inserting a element into the list. We can now declare a function that checks that two sets do not share any elements. It is the intuitive recursive function that checks each member of the first list against the second list.

```
int SET_Disjoint(SET_set set1, SET_set set2){
  if (set1 != NULL){
      if (SET_IsMember(set2,set1->data)){
          return FALSE;
        }
      else{
          return SET_Disjoint((set1->next),set2);
        }
    }
  else{
      return TRUE;
    }
}
```

Using this function and creating the sets of heap allocated variables as we recurse the abstract syntax tree, we can perform the necessary linearity checks.

4.6 Translation

As described in section 2.2 the C code produced is split into three distinct sections, each stored in a string,

```
char *types;
char *prototypes;
char *functions;
 A_type_list produced_already = NULL;
```

As we encounter types and produce code for them we enter them into the above list, however we also recursively produce the code for the sub types, and insert them into our type string first.

```
char *TRANS_recursive_typecode(A_type type);
```

This will guarantee that C type code is produced in the correct order. see appendix A.20 for details of code that produces types, as well the other translation functions.

As we translate a function the variables storing intermediate results must must be declared at the beginning, so we must store these in a separate string,

```
char *declarations;
```

when we have finished translating the body of a function and adding to this string, we can build the string representing of the function out of it component parts and clear this global string.

The translation phase follows as similar design to the semantic analysis, traversing the abstract syntax tree. However it is only concerned with mapping `symbols` to their C code representation, so only mains one symbol table,

```
S_table variables;
```

Due to the recursive nature of the creation of the C code, many smaller strings have to be joined together. Further, as strings are static, we are required to create a bigger string and copy into it whenever wishing to join strings together. To make this process easier the following function was written,

```
char *concat_with_free(char *a, char *b)
{
  char *temp;
  temp = calloc(strlen(a) + strlen(b) + 1, sizeof(char));
  strcpy(temp, a);
  strcat(temp, b);
  free(a);
  free(b);
  return temp;
}
```

4.6.1 Variable Naming

To create the unique string representing a valid C identifier a global counter is used,

```
static int VAR_counter = 0;
```

This will allow `sizeof(int)` number of unique identifiers to be created, which although finite, is likely to be sufficient, particularly as it can be reset for each function defined,

```
void VAR_reset(void)
```

To create a valid C identifier from this counter it must first be converted to a string, and as all identifiers must begin with a letter, a character must then be prefixed to it. This is implemented by the function,

```
char *VAR_fresh(void){
  char *temp;
  temp = calloc(20, sizeof(char));
  sprintf(temp, "x\%d",VAR_counter);
  VAR_counter++;
  return temp;
}
```

when a new identifier is require a call to this function will return a previously unused name of the for x1, x2, x3, ... etc

4.7 Example

Below can be seen the C code produced by the compiler after the successful translation of the Reverse function given in source language form in section 3.4. Note the three distinct sections mentioned above, separated by comments, and also the relatively poor formatting of the functions, this is discussed in section 6.1.

It should be further noted that while this is valid C code, it will not compile as it stands as there is no `main` function.

Reverse:

```
/* TYPE DEFINITIONS */
typedef enum {INL, INR, NIL, CONS, LEAF, NODE} kind_t;

typedef void *d;

typedef int i;

typedef struct{
```

```
kind_t kind;
i hd;
void *tl;
} li;

li li_nil(void)
{
li res;
res.kind = NIL;
return res;
}

li li_cons(d d, i hd, li tl)
{
li res;
res.kind = CONS;
res.hd = hd;
*(li *)d = tl;
res.tl = d;
return res;
}

typedef struct{
kind_t kind;
i hd;
li tl;
d d;
}li_destr_t;

li_destr_t li_destr(li l)
{
li_destr_t res;
res.kind = l.kind;
if(res.kind == CONS)
{
res.hd = l.hd;
res.d = (void *)l.tl;
res.tl = *(li *)l.tl;
}
return res;
}


/* FUNCTION PROTOTYPES */
li rev_aux(li, li);

li reverse(li);


/* FUNCTION DEFINITIONS */
li rev_aux(li l, li acc)
```

```
{
li_destr_t x0;

return (x0=li_destr(l), x0.kind == NIL ? acc :
         rev_aux(x0.tl, li_cons(x0.d, x0.hd, acc)));
}

li reverse(li l)
{

return rev_aux(l, li_nil());
}
```

5. Analysis

5.1 Testing and Functionality

As the compiler was designed in a modular manner, with clear principals, testing its functionality was greatly simplified. Each stage has a clean interface, which allows internal errors to be detected. They either resulted in a incorrect interface that could not connected to the other stages, or the interface did not produce the required output. This meant error could traced to a particular module.

The use of error messages and a pretty printer helped with debugging each phase. Careful chose of input would produce error messages that could used to locate an error. The testing of the compiler was carried as each phase was completed. The method used was simple and though not rigorous was sufficient to high light problems with the development of the system.

The compiler had a front-end driver function wrapped around it that allowed input to be read from files, and phases turned on or off. Test input was constructed for each stage of the compiler, with representative correct input and what was considered probable error producing input.

The only problem with the completed compiler that causes error and has been detected using this method was a test left out of semantic analysis phase that can result in invalid C code being produced. see section 6.1 for details.

Certain other conditions may cause the compiler to crash, for example running out of intermediate identifiers, however these were design decisions made to ease implementation of a working system and are documented in this paper.

The code for the front end is given in Appendix A.7.

Details of how to execute the compiler using its front end shackle and where to obtain sample test files can be found in the introduction to the Appendices A.

5.2 Comparison with other Systems

Comparing the compiler to other system was investigated, however no concrete tests could be carried out, so there is no data with which to form an opinion.

To compare the memory usage with other traditional functional systems,such as OCAML [7] and New Jersey SML [12] it seemed appropriate to use common examples that could be represented in all languages. It was decided that breadth first traversal of a large binary tree was appropriate. The idea was to create large tree of say depth 20, and then call the breadth function mentioned earlier. To construct the tree the follow definitions were made,

```
d new(){
  return (d) malloc(sizeof(lti));
}

void disp(d d){
  free((lti*)d);
}

ti make(int depth){
  if(depth == 0)
    {
      return ti_leaf(7);
    }
  else
    {
      return ti_node(new(),new(),7,make(depth-1),make(depth-1));
    }
}

int main(void){
  breadth(lti_cons(new(),make(20),lti_nil()));
  return 0;
}
```

This code can the be compiled and linked with a memory profiler such as Mpatrol [11] of eFence [10]. When it is executed the profiler produces a summary file showing, amongst other things, the maximum size of the heap. The figures reported include a large overhead of unknown size caused by the profiler itself, so an exact measure cannot be gleamed. (though it can be calculated from the code).

Comparison was further hampered by lack of memory profilers for any of the functional languages investigated. Approximate figures can be determined by the use to the Unix command 'top', however again figures were not particularly meaningful.

Despite the lack of performance measures, this compiler has the advantage over other traditional compilers, in that it can place a bound on the amount of heap space used.

6. Further Work

Although a working compiler has been constructed, it was far from complete. It is not for practical applications and needs some work to make it fully functional. If it is to have industrial application then the system must be expanded with new functionality to increase its usefulness. As it stands the compiler produces valid C code, but this must be augmented with hand written code to produce any kind of relevant results.

6.1 Omissions and Corrections

The current compiler fails to check that parameter identifiers listed in a function definition are unique. This is not serious, as most programmer use unique parameter identifiers intuitively. However, if they are not unique the C code that is produced will not compile. This can be corrected by introducing another test in the semantic analysis phase. As the list of parameter identifiers is created for each definition, the identifier about to be added, should checked against the existing list and rejected if found to already exist. This check could be easily incorporated into the current system, it's absence was simply caused by oversight.

It is perhaps ironic that a compiler that produces code that guarantees to use heap space linearly has potential space leaks itself. As mentioned in section 3.3 certain data structure are not deallocated when finished with, this could result in the system running out of memory when compiling (very) large program. Again this is not of too much concern, as it will not cause the compiler to crash because of prudent use of the `checked_malloc()` function. However, an industrial strength compiler should have a more sophisticated memory management scheme. with a little work, deconstructor functions could be implemented for all data structures used in the system.

The C code string that is produced is not formatted in the traditional C style, therefore making reading and understanding it relatively difficult. This is purely an aesthetic consideration but well formatted code has it's advantages As the system is new, programmers approaching it for the first time may compare the C code produced to the functional language it was derived from to aid there understanding. Further, as currently the code must be used with programmer generated code it will often be referred to during creation of these extra function. Consequently, well formatted code would advantageous. This can be done by inserting the appropriate 'white space' in the the output of translation stage, no new features would need to be added to the compiler.

To increase the robustness of the system and help a programmer debug his code, the compiler could implement some form of error recovery. This would allow the system to continue checking source code after a error has occurred. If error productions were included in the grammar for the language then this could be help achieved. For example, a function with a parsing error could be skipped and the process continue with next one. This may result in a type checking error if a later function calls on the skipped one, but system would still be useful.

The performance of the system could be improved by re-implementing the `sets` and `table` data structures. The current system uses an acceptable but inefficient list representation for sets. As the system was designed in a modular way a new more efficient implantation can easily be substituted. It need simply meet the function definitions given in the header file. However, the present method gives acceptable results.

6.2 Extensions

The present system does not allow for the creation of values of type $\diamond$, so concreting constants of heap allocated types cannot used. This restricts the system's potential uses rather a lot, dynamic allocation would increase the range of functions that could be defined. However this can be achieved by the introduction of two new functions into the source language,

```
new()diamond

disp(diamond)int;
```

these would be translated into C functions,

```
d new(){
        return malloc(sizeof(T));
}

i disp(d d){
        free d;
        return 0;
}
```

T will have to be big enough to hold the largest heap allocated type. This could be achieved by creating a union of the all types used within a program. However it need only be declared to be equal to the biggest, and this can be determined by the compiler as it constructs the type section of target code as discussed in section 4.6. Implementing this will make the single biggest increase in the usefulness of the current system.

A minor addition to language would be the inclusion of more binary operators and more importantly, unary negation. Again, this would add to the expressive power of the language.

The addition of a type inference phase to the compiler would allow the type that is currently required in `nil` expressions to be dropped. It would also alleviate the need for the programmer to explicitly manipulate the $\diamond$ type. This would make the programmers task easier as it is often difficult to determine this type. Implementing a type inference system phase would ease the implementation of a polymorphic version of the language. The method for incorporating such a inference system into our compiler is given in some detail in [2].

An other minor addition, this time to the compiler, would be to move comments given in the source language into the target code. This helps maintain the accessibility of code, even after translation. This would take little effort to implement.

Several other extensions to the language that would increase its effectiveness as a general purpose programming language, including polymorphism and higher order functions, are discussed in [5].

7. Conclusions

The aim of this paper was to show that a compiler could be implemented for the functional language presented in [5], given it's unique properties. These properties, linear typing and explicit control of heap space through the use of a resource type $\diamond$, mean that in-place update of heap allocated data structures can be achieved in a functional style. The compiler was to translate this language into `malloc`()-free C code according to the given rules.

This has been demonstrated and a working system developed. The complier produced can correctly translate the functional language into valid C which can be used as part of a larger program.

The unique features of the functional language results in the compiler having two novel characteristics.

- To insure linearity the compiler must check that typing contexts are disjoint. This requires maintaining sets of variables of heap allocated types used within context and being able to test that two given contexts do not have any variables in common. This was implement using a sets abstract data type.

- As C does not have any constructs that bind variables in a similar way to the `match` construct in the functional language, the code produced by the compiler contained a lot of function-global variables to hold intermediate results. This required the compiler to generate many unique variable identifiers. This was achieved by maintaining a global counter that was used to produce identifiers of the form x1, x2 ,x3...etc.

As the code produced by the compiler can be executed in a heap of statically determinable size the system may have possible applications in environments where memory is at a premium, such as programming for embedded system or computation over the Internet.

However the compiler is not of 'industrial strength' as it stands. It functionality is basic, which limits its practical use as an implementation system. If it is extended in the ways mention in chapter 6, all of which a reasonably straightforward, then it may yet be a fully-fledge system for development and realize the potential for automated strict heap space management.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, first edition, 1986.

[2] Andrew W. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, first edition, 1998.

[3] Richard Stallman Charles Donnelly. Bison-the yacc compatible parse generator, v1.16, 1991.

[4] Martin Hofmann. Linear types and non-size-increasing polynomial time computation. *LICS*, 1999.

[5] Martin Hofmann. A type system for bouned space and functional in-place update. In G. Smolka, editor, *Programming Laguages and Systems*, pages 165–179. Springer LNCS, 2000.

[6] Al Kelly and Ira Pohl. *A book on C*. Benjamin/Cummings, third edition, 1995.

[7] Xavier Leroy. The objective caml system, documentation and user's guide. http://pauillac.inria.fr/ocaml/htmlman, 1999.

[8] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, second edition, 1996.

[9] V. Paxson. Flex-fast lexical analyzer generator. Lawrence Berkeley Laboratory, Berkeley, CA, 1995. ftp://ftp:ee.lbl.gov/flex-2.5.3.tar.gz.

[10] Bruce Perens. Electric fence, malloc debugger. bruce@perens.com, 1993.

[11] Graeme S. Roy. Mpatrol v1.1, memory profiler. http://www.cbmamiga.demon.co.uk/mpatrol/, 2000.

[12] various. Standard ml of new jersey documentation. ftp://ftp.research.bell-labs.com/dist/smlnj/doc/SMLNJ/smlnj.tar.Z, 1999.

Appendix A. Code Listing

Below are all the files that make up the complete compiler. They can also be found at *http://www.dcs.ed.ac.uk/home/nbb/complier.tar.zip* as a tarred and zipped file. To extract the files use the following commands;

```
> gunzip compiler.tar.zip
> tar -xvf compiler.tar
```

To build the compiler issue the following command within the compiler directory;

```
> make compiler
```

The compiler has the follow command structure;

compiler [**-parse**|**-check**] *file*

The compiler will produce the C code translation of the given source code file on its standard output, with allow error and status messages going to standard error.

file	is the name of the source code file.
-parse	the compiler will parse the source code for errors and output a pretty printed form of the abstract syntax.
-check	the compiler will parse and type check the source code, but will not produce the translated C code.

Some sample source code files are included with the compiler.

A.1 abstractsyntax.h

```
/* type defintions */

typedef int A_pos;

typedef struct A_program_ *A_program;
typedef struct A_signature_definition_ *A_signature_definition;
typedef struct A_signature_ *A_signature;
typedef struct A_definition_ *A_definition;
typedef struct A_variable_ *A_variable;
typedef struct A_type_ *A_type;
typedef struct A_expression_ *A_expression;

typedef struct A_sig_def_list_ *A_sig_def_list;
typedef struct A_variable_list_ *A_variable_list;
typedef struct A_type_list_ *A_type_list;
typedef struct A_expression_list_ *A_expression_list;

typedef enum {A_plusOp, A_minusOp, A_multiplyOp, A_divideOp, A_lessOp,
A_greaterOp, A_sameOp} A_oper;

struct A_program_
{
  A_sig_def_list sig_def_list;
};

struct A_signature_definition_
{
  enum
  {
    A_sig, A_def
  }kind;
  union
  {
    A_signature sig;
    A_definition def;
  }u;
```

```
};

struct A_signature_
{
  A_pos pos;
  S_symbol name;
  A_type_list parameter_type_list;
  A_type result;
};

struct A_definition_
{
  A_pos pos;
  S_symbol name;
  A_variable_list parameter_list;
  A_expression body;
};

struct A_variable_
{
  A_pos pos;
  S_symbol name;
  A_type type;
};

struct A_type_
{
  enum
  {
    A_int_Type, A_diamond_Type, A_list_Type, A_tree_Type, A_pair_Type
  }kind;
  A_pos pos;
  union
  {
    // int type needs nothing
    // diamond type needs nothing
    A_type list_type;
    A_type tree_type;
    struct { A_type left; A_type right;} pair;
  }u;
};

struct A_expression_
{
  enum
  {
    A_variable_Exp, A_number_Exp, A_operator_Exp, A_if_Exp, A_pair_Exp,
A_nil_Exp,
    A_cons_Exp, A_leaf_Exp, A_node_Exp, A_function_call_Exp,
    A_match_list_Exp, A_match_pair_Exp, A_match_tree_Exp
  }kind;
  A_type type;
  A_pos pos;
  union
  {
    A_variable variable;
    int intt;
    struct {A_oper oper; A_expression left; A_expression right;} operator;
    struct {A_expression test; A_expression then; A_expression elsee;} iff;
    struct {A_expression left; A_expression right;} pair;
    A_type nilof;
    struct {A_expression a,b,c;} cons;
    A_expression leaf;
    struct {A_expression a,b,c,d,e;} node;
    struct {S_symbol name; A_expression_list arguments;} function_call;
    struct
    {
      A_expression a;
      A_expression b;
      A_variable c,d,e;
      A_expression f;
    } match_list;
    struct
    {
      A_expression a;
      A_variable b;
      A_variable c;
      A_expression d;
    } match_pair;
    struct
    {
      A_expression a;
      A_variable b;
      A_expression c;
      A_variable d,e,f,g,h;
      A_expression i;
    } match_tree;
  } u;
```

```
};

/* linked lists */

struct A_sig_def_list_
{
  A_signature_definition head;
  A_sig_def_list tail;
};

struct A_variable_list_
{
  A_variable head;
  A_variable_list tail;
};

struct A_type_list_
{
  A_type head;
  A_type_list tail;
};

struct A_expression_list_
{
  A_expression head;
  A_expression_list tail;
};

/* constructor functions */

A_program A_Program(A_sig_def_list sig_def_list);
A_sig_def_list A_Sig_Def_List(A_signature_definition head, A_sig_def_list tail);
A_signature_definition A_Sig(A_signature signature);
A_signature_definition A_Def(A_definition definition);
A_signature A_Signature(A_pos pos, S_symbol name, A_type_list params_types,
A_type result);
A_definition A_Definition(A_pos pos, S_symbol name, A_variable_list parms,
A_expression body);
A_variable_list A_Variable_List(A_variable head, A_variable_list tail);
A_variable A_Variable(A_pos pos, S_symbol name, A_type type);
A_type_list A_Type_List(A_type head, A_type_list tail);
A_type A_Int_Type(A_pos pos);
A_type A_Diamond_Type(A_pos pos);
A_type A_List_Type(A_pos pos, A_type list_type);
A_type A_Tree_Type(A_pos pos, A_type tree_type);
A_type A_Pair_Type(A_pos pos, A_type left, A_type right);
A_expression_list A_Expression_List(A_expression head, A_expression_list tail);
A_expression A_Variable_Exp(A_type type, A_pos pos, A_variable variable);
A_expression A_Number_Exp(A_type type, A_pos pos, int i);
A_expression A_Operator_Exp(A_type type, A_pos pos, A_oper oper, A_expression
left, A_expression right);
A_expression A_If_Exp(A_type type, A_pos pos, A_expression test, A_expression
then, A_expression elsee);
A_expression A_Pair_Exp(A_type type, A_pos pos, A_expression left, A_expression
right);
A_expression A_Nil_Exp(A_type type, A_pos pos, A_type nilof);
A_expression A_Cons_Exp(A_type type, A_pos pos, A_expression a, A_expression b,
A_expression c);
A_expression A_Leaf_Exp(A_type type, A_pos pos, A_expression leaf);
A_expression A_Node_Exp(A_type type, A_pos pos, A_expression a, A_expression b,
A_expression c, A_expression d, A_expression e);
A_expression A_Function_Call_Exp(A_type type, A_pos pos, S_symbol name,
A_expression_list arguments);
A_expression A_Match_List_Exp(A_type type, A_pos pos, A_expression a,
A_expression b, A_variable c, A_variable d, A_variable e, A_expression f);
A_expression A_Match_Pair_Exp(A_type type, A_pos pos,  A_expression a,
A_variable b, A_variable c, A_expression d);
A_expression A_Match_Tree_Exp(A_type type, A_pos pos,  A_expression a,
A_variable b, A_expression c, A_variable d, A_variable e, A_variable f,
A_variable g, A_variable h, A_expression i);
```

A.2 abstractsyntax.c

```
#include "util.h"
#include "symbol.h"
#include "abstractsyntax.h"


A_program A_Program(A_sig_def_list sig_def_list)
{
  A_program p = (A_program)checked_malloc(sizeof(*p));
  p-> sig_def_list = sig_def_list;
  return p;
```

```
}

A_sig_def_list A_Sig_Def_List(A_signature_definition head, A_sig_def_list tail)
{
  A_sig_def_list p = (A_sig_def_list)checked_malloc(sizeof(*p));
  p->head = head;
  p->tail = tail;
  return p;
}

A_signature_definition A_Sig(A_signature signature)
{
  A_signature_definition p = (A_signature_definition)checked_malloc(sizeof(*p));
  p->kind = A_sig;
  p->u.sig = signature;
  return p;
}

A_signature_definition A_Def(A_definition definition)
{
  A_signature_definition p = (A_signature_definition)checked_malloc(sizeof(*p));
  p->kind = A_def;
  p->u.def = definition;
  return p;
}

A_signature A_Signature(A_pos pos, S_symbol name, A_type_list params_types,
A_type result)
{
  A_signature p = (A_signature)checked_malloc(sizeof(*p));
  p->pos = pos;
  p->name = name;
  p->parameter_type_list = params_types;
  p->result = result;
  return p;
}

A_definition A_Definition(A_pos pos, S_symbol name, A_variable_list parms,
A_expression body)
{
  A_definition p = (A_definition)checked_malloc(sizeof(*p));
  p->pos = pos;
  p->name = name;
  p->parameter_list = parms;
  p->body = body;
  return p;
}

A_variable_list A_Variable_List(A_variable head, A_variable_list tail)
{
  A_variable_list p = (A_variable_list)checked_malloc(sizeof(*p));
  p->head = head;
  p->tail = tail;
  return p;
}

A_variable A_Variable(A_pos pos, S_symbol name, A_type type)
{
  A_variable p = (A_variable)checked_malloc(sizeof(*p));
  p->pos = pos;
  p->name = name;
  p->type = type;
  return p;
}

A_type_list A_Type_List(A_type head, A_type_list tail)
{
  A_type_list p = (A_type_list)checked_malloc(sizeof(*p));
  p->head = head;
  p->tail = tail;
  return p;
}

A_type A_Int_Type(A_pos pos)
{
  A_type p = (A_type)checked_malloc(sizeof(*p));
  p->kind = A_int_Type;
  p->pos = pos;
  return p;
}

A_type A_Diamond_Type(A_pos pos)
{
  A_type p = (A_type)checked_malloc(sizeof(*p));
  p->kind = A_diamond_Type;
  p->pos = pos;
  return p;
}
```

```
A_type A_List_Type(A_pos pos, A_type list_type)
{
  A_type p = (A_type)checked_malloc(sizeof(*p));
  p->kind = A_list_Type;
  p->pos = pos;
  p->u.list_type = list_type;
  return p;
}

A_type A_Tree_Type(A_pos pos, A_type tree_type)
{
  A_type p = (A_type)checked_malloc(sizeof(*p));
  p->kind = A_tree_Type;
  p->pos = pos;
  p->u.tree_type = tree_type;
  return p;
}

A_type A_Pair_Type(A_pos pos, A_type left, A_type right)
{
  A_type p = (A_type)checked_malloc(sizeof(*p));
  p->kind = A_pair_Type;
  p->pos = pos;
  p->u.pair.left = left;
  p->u.pair.right = right;
  return p;
}

A_expression_list A_Expression_List(A_expression head, A_expression_list tail)
{
  A_expression_list p = (A_expression_list)checked_malloc(sizeof(*p));
  p->head = head;
  p->tail = tail;
  return p;
}

A_expression A_Variable_Exp(A_type type, A_pos pos, A_variable variable)
{
  A_expression p = (A_expression)checked_malloc(sizeof(*p));
  p->type = type;
  p->kind = A_variable_Exp;
  p->pos = pos;
  p->u.variable = variable;
  return p;
}

A_expression A_Number_Exp(A_type type, A_pos pos, int i)
{
  A_expression p = (A_expression)checked_malloc(sizeof(*p));
  p->type = type;
  p->kind = A_number_Exp;
  p->pos = pos;
  p->u.intt = i;
  return p;
}

A_expression A_Operator_Exp(A_type type, A_pos pos, A_oper oper, A_expression
left, A_expression right)
{
  A_expression p = (A_expression)checked_malloc(sizeof(*p));
  p->type = type;
  p->kind = A_operator_Exp;
  p->pos = pos;
  p->u.operator.oper = oper;
  p->u.operator.left = left;
  p->u.operator.right = right;
  return p;
}

A_expression A_If_Exp(A_type type, A_pos pos, A_expression test, A_expression
then, A_expression elsee)
{
  A_expression p = (A_expression)checked_malloc(sizeof(*p));
  p->type = type;
  p->kind = A_if_Exp;
  p->pos = pos;
  p->u.iff.test = test;
  p->u.iff.then = then;
  p->u.iff.elsee = elsee;
  return p;
}

A_expression A_Pair_Exp(A_type type, A_pos pos, A_expression left, A_expression
right)
{
  A_expression p = (A_expression)checked_malloc(sizeof(*p));
  p->type = type;
```

```
  p->kind = A_pair_Exp;
  p->pos = pos;
  p->u.pair.left = left;
  p->u.pair.right = right;
  return p;
}

A_expression A_Nil_Exp(A_type type, A_pos pos, A_type nilof)
{
  A_expression p = (A_expression)checked_malloc(sizeof(*p));
  p->type = type;
  p->kind = A_nil_Exp;
  p->pos = pos;
  p->u.nilof = nilof;
  return p;
}

A_expression A_Cons_Exp(A_type type, A_pos pos, A_expression a, A_expression b,
A_expression c)
{
  A_expression p = (A_expression)checked_malloc(sizeof(*p));
  p->type = type;
  p->kind = A_cons_Exp;
  p->pos = pos;
  p->u.cons.a = a;
  p->u.cons.b = b;
  p->u.cons.c = c;
  return p;
}

A_expression A_Leaf_Exp(A_type type, A_pos pos, A_expression leaf)
{
  A_expression p = (A_expression)checked_malloc(sizeof(*p));
  p->type = type;
  p->kind = A_leaf_Exp;
  p->pos = pos;
  p->u.leaf = leaf;
  return p;
}

A_expression A_Node_Exp(A_type type, A_pos pos, A_expression a, A_expression b,
A_expression c, A_expression d, A_expression e)
{
  A_expression p = (A_expression)checked_malloc(sizeof(*p));
  p->type = type;
  p->kind = A_node_Exp;
  p->pos = pos;
  p->u.node.a = a;
  p->u.node.b = b;
  p->u.node.c = c;
  p->u.node.d = d;
  p->u.node.e = e;
  return p;
}

A_expression A_Function_Call_Exp(A_type type, A_pos pos, S_symbol name,
A_expression_list arguments)
{
  A_expression p = (A_expression)checked_malloc(sizeof(*p));
  p->type = type;
  p->kind = A_function_call_Exp;
  p->pos = pos;
  p->u.function_call.name = name;
  p->u.function_call.arguments = arguments;
  return p;
}

A_expression A_Match_List_Exp(A_type type, A_pos pos, A_expression a,
A_expression b, A_variable c, A_variable d, A_variable e, A_expression f)
{
  A_expression p = (A_expression)checked_malloc(sizeof(*p));
  p->type = type;
  p->kind = A_match_list_Exp;
  p->pos = pos;
  p->u.match_list.a = a;
  p->u.match_list.b = b;
  p->u.match_list.c = c;
  p->u.match_list.d = d;
  p->u.match_list.e = e;
  p->u.match_list.f = f;
  return p;
}

A_expression A_Match_Pair_Exp(A_type type, A_pos pos,  A_expression a,
A_variable b, A_variable c, A_expression d)
{
  A_expression p = (A_expression)checked_malloc(sizeof(*p));
  p->type = type;
```

```
  p->kind = A_match_pair_Exp;
  p->pos = pos;
  p->u.match_pair.a = a;
  p->u.match_pair.b = b;
  p->u.match_pair.c = c;
  p->u.match_pair.d = d;
  return p;
}

A_expression A_Match_Tree_Exp(A_type type, A_pos pos,  A_expression a,
A_variable b, A_expression c, A_variable d, A_variable e, A_variable f,
A_variable g, A_variable h, A_expression i)
{
  A_expression p = (A_expression)checked_malloc(sizeof(*p));
  p->type = type;
  p->kind = A_match_tree_Exp;
  p->pos = pos;
  p->u.match_tree.a = a;
  p->u.match_tree.b = b;
  p->u.match_tree.c = c;
  p->u.match_tree.d = d;
  p->u.match_tree.e = e;
  p->u.match_tree.f = f;
  p->u.match_tree.g = g;
  p->u.match_tree.h = h;
  p->u.match_tree.i = i;
  return p;
}
```

A.3 errormsg.h

```
extern bool EM_anyErrors;

void EM_newline(void);

extern int EM_tokPos;

void EM_error(int, string,...);
void EM_impossible(string,...);
void EM_reset(string filename);
```

A.4 errormsg.c

```
/*
 * errormsg.c - functions used in all phases of the compiler to give
 *              error messages about the Tiger program.
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <stdarg.h>
#include "util.h"
#include "errormsg.h"


bool anyErrors= FALSE;

static string fileName = "";

static int lineNum = 1;

int EM_tokPos=0;

extern FILE *yyin;

typedef struct intList {int i; struct intList *rest;} *IntList;

static IntList intList(int i, IntList rest)
{IntList l= checked_malloc(sizeof *l);
 l->i=i; l->rest=rest;
 return l;
}

static IntList linePos=NULL;

void EM_newline(void)
{lineNum++;
```

```
 linePos = intList(EM_tokPos, linePos);
}

void EM_error(int pos, char *message,...)
{va_list ap;
 IntList lines = linePos;
 int num=lineNum;


  anyErrors=TRUE;
  while (lines && lines->i >= pos)
       {lines=lines->rest; num--;}

  if (fileName) fprintf(stderr,"%s:",fileName);
  if (lines) fprintf(stderr,"line %d, position %d: ", num, pos-lines->i);
  va_start(ap,message);
  vfprintf(stderr, message, ap);
  va_end(ap);
  fprintf(stderr,"\n");
  exit(0);
}

void EM_reset(string fname)
{
 anyErrors=FALSE; fileName=fname; lineNum=1;
 linePos=intList(0,NULL);
 yyin = fopen(fname,"r");
 if (!yyin) {EM_error(0,"cannot open"); exit(1);}
}
```

A.5 fun.grm

```
%{
#include <string.h>
#include <stdio.h>
#include "util.h"
#include "symbol.h"
#include "errormsg.h"
#include "abstractsyntax.h"

#define YYERROR_VERBOSE 1

int yylex(void);

A_program abstractsyntax_root;

void yyerror( char *string)
{
EM_error(EM_tokPos, "%s", string);
}

%}

%union {
        int pos;
int ival;
string sval;
A_program program;
A_sig_def_list sig_def_list;
A_signature_definition signature_definition;
A_signature signature;
A_definition definition;
A_variable_list variable_list;
A_variable variable;
A_type_list type_list;
A_type type;
A_expression_list expression_list;
A_expression expression;
        }

%token <sval>ID
%token <ival>NUMBER
%token FUN COLON
%token COMMA
%token LEFT_PAR RIGHT_PAR LEFT_BRAC RIGHT_BRAC
%token INT NIL DIAMOND LIST TREE
%token EQUAL
%right MATCH WITH GOESTO BAR
%right IF THEN ELSE
%left CONS NODE LEAF
%left LESS_THAN GREATER_THAN SAME_AS
%left PLUS MINUS
```

```
%left MULTIPLY DIVIDE

%type <program> program
%type <sig_def_list> sig_def_list
%type <signature_definition> signature_definition
%type <signature> signature
%type <definition> definition
%type <type_list> parameter_type_list
%type <type> parameter_type type_specifier
%type <variable_list> parameter_list
%type <variable> parameter variable
%type <expression> expression binop match cons leaf node function_call
%type <expression_list> expression_list

%start program

%%

program : sig_def_list  {abstractsyntax_root = A_Program($1);}
        |  {abstractsyntax_root = NULL;}

sig_def_list: signature_definition sig_def_list  {$$ = A_Sig_Def_List($1, $2);}
             | signature_definition  {$$ = A_Sig_Def_List($1, NULL);}

signature_definition: signature  {$$ = A_Sig($1);}
                    | definition  {$$ = A_Def($1);}

signature: ID COLON LEFT_PAR parameter_type_list RIGHT_PAR type_specifier
{$$=A_Signature(EM_tokPos,S_Symbol($1),$4,$6);}
         | ID COLON LEFT_PAR RIGHT_PAR type_specifier
{$$=A_Signature(EM_tokPos,S_Symbol($1),NULL,$5);}

parameter_type_list: parameter_type COMMA parameter_type_list
{$$=A_Type_List($1,$3);}
                   | parameter_type {$$=A_Type_List($1,NULL);}

parameter_type: type_specifier {$$=$1;}

type_specifier: INT {$$=A_Int_Type(EM_tokPos);}
              | LIST LEFT_PAR type_specifier RIGHT_PAR
{$$=A_List_Type(EM_tokPos,$3);}
              | TREE LEFT_PAR type_specifier RIGHT_PAR
{$$=A_Tree_Type(EM_tokPos,$3);}
              | LEFT_BRAC type_specifier COMMA type_specifier RIGHT_BRAC
{$$=A_Pair_Type(EM_tokPos,$2,$4);}
              | DIAMOND {$$=A_Diamond_Type(EM_tokPos);}

definition: FUN ID LEFT_PAR parameter_list RIGHT_PAR EQUAL expression
{$$=A_Definition(EM_tokPos,S_Symbol($2),$4,$7);}
          | FUN ID LEFT_PAR RIGHT_PAR EQUAL expression
{$$=A_Definition(EM_tokPos,S_Symbol($2),NULL,$6);}

parameter_list: parameter COMMA parameter_list {$$=A_Variable_List($1,$3);}
              | parameter {$$=A_Variable_List($1,NULL);}

parameter: variable {$$=$1;}

variable: ID {$$=A_Variable(EM_tokPos, S_Symbol($1),NULL);}

expression: variable {$$=A_Variable_Exp(NULL, EM_tokPos,$1);}
          | NUMBER {$$=A_Number_Exp(NULL, EM_tokPos,$1);}
          | LEFT_PAR expression RIGHT_PAR {$$=$2;}
          | binop {$$=$1;}
          | IF expression THEN expression ELSE expression {$$=A_If_Exp(NULL,
EM_tokPos,$2,$4,$6);}
          | match {$$=$1;}
          | LEFT_BRAC expression COMMA expression RIGHT_BRAC
{$$=A_Pair_Exp(NULL, EM_tokPos,$2,$4);}
          | NIL LEFT_PAR type_specifier RIGHT_PAR {$$=A_Nil_Exp($3,
EM_tokPos,$3);}
          | cons {$$=$1;}
          | leaf {$$=$1;}
          | node {$$=$1;}
          | function_call {$$=$1;}

binop: expression LESS_THAN expression {$$=A_Operator_Exp(NULL,
EM_tokPos,A_lessOp,$1,$3);}
     | expression GREATER_THAN expression {$$=A_Operator_Exp(NULL,
EM_tokPos,A_greaterOp,$1,$3);}
     | expression SAME_AS expression {$$=A_Operator_Exp(NULL,
EM_tokPos,A_sameOp,$1,$3);}
     | expression PLUS expression {$$=A_Operator_Exp(NULL,
EM_tokPos,A_plusOp,$1,$3);}
     | expression MINUS expression {$$=A_Operator_Exp(NULL,
EM_tokPos,A_minusOp,$1,$3);}
     | expression MULTIPLY expression {$$=A_Operator_Exp(NULL,
EM_tokPos,A_multiplyOp,$1,$3);}
     | expression DIVIDE expression {$$=A_Operator_Exp(NULL,
```

```
EM_tokPos,A_divideOp,$1,$3);}

match: MATCH expression WITH LEAF LEFT_PAR variable RIGHT_PAR GOESTO expression
BAR NODE LEFT_PAR variable COMMA variable COMMA variable COMMA variable COMMA
variable RIGHT_PAR GOESTO expression {$$=A_Match_Tree_Exp(NULL,
EM_tokPos,$2,$6,$9,$13,$15,$17,$19,$21,$24);}
          | MATCH expression WITH NIL GOESTO expression BAR CONS LEFT_PAR
variable COMMA variable COMMA variable RIGHT_PAR GOESTO expression
{$$=A_Match_List_Exp(NULL, EM_tokPos,$2,$6,$10,$12,$14,$17);}
          | MATCH expression WITH LEFT_BRAC variable COMMA variable RIGHT_BRAC
GOESTO expression {$$=A_Match_Pair_Exp(NULL, EM_tokPos,$2,$5,$7,$10);}

cons: CONS LEFT_PAR expression COMMA expression COMMA expression RIGHT_PAR
{$$=A_Cons_Exp(NULL, EM_tokPos,$3,$5,$7);}

leaf: LEAF LEFT_PAR expression RIGHT_PAR {$$=A_Leaf_Exp(NULL, EM_tokPos,$3);}

node: NODE LEFT_PAR expression COMMA expression COMMA expression COMMA
expression COMMA expression RIGHT_PAR {$$=A_Node_Exp(NULL,
EM_tokPos,$3,$5,$7,$9,$11);}

function_call: ID LEFT_PAR expression_list RIGHT_PAR
{$$=A_Function_Call_Exp(NULL, EM_tokPos,S_Symbol($1),$3);}
             | ID LEFT_PAR RIGHT_PAR {$$=A_Function_Call_Exp(NULL,
EM_tokPos,S_Symbol($1),NULL);}

expression_list: expression {$$=A_Expression_List($1,NULL);}
               | expression COMMA expression_list {$$=A_Expression_List($1,$3);}
```

A.6 fun.lex

```
%{

#include <string.h>
#include <stdlib.h>
#include "util.h"
#include "symbol.h"
#include "abstractsyntax.h"
#include "fun.grm.tab.h"
#include "errormsg.h"

int charPos = 1;

int yywrap(void)
  {
    charPos = 1;
    return 1;
  }

void adjust(void)
  {
    EM_tokPos=charPos;
    charPos+=yyleng;
  }

%}

%x COMMENT
%%

" " {adjust();continue;}
"\n" {adjust(); EM_newline(); continue;}
"\t" {adjust(); continue;}

"/*" {adjust(); BEGIN COMMENT;}
"," {adjust(); return COMMA;}
"fun" {adjust(); return FUN;}
":" {adjust(); return COLON;}
"list" {adjust(); return LIST;}
"tree" {adjust(); return TREE;}
"[" {adjust(); return LEFT_BRAC;}
"]" {adjust(); return RIGHT_BRAC;}
"(" {adjust(); return LEFT_PAR;}
")" {adjust(); return RIGHT_PAR;}
"int" {adjust(); return INT;}
"match" {adjust(); return MATCH;}
"with" {adjust(); return WITH;}
"if" {adjust(); return IF;}
"else" {adjust(); return ELSE;}
"then" {adjust(); return THEN;}
"nil" {adjust(); return NIL;}
"cons" {adjust(); return CONS;}
"leaf" {adjust(); return LEAF;}
```

```
"node" {adjust(); return NODE;}
"|" {adjust(); return BAR;}
"=>" {adjust(); return GOESTO;}
"=" {adjust(); return EQUAL;}
"+" {adjust(); return PLUS;}
"-" {adjust(); return MINUS;}
"*" {adjust(); return MULTIPLY;}
"/" {adjust(); return DIVIDE;}
">" {adjust(); return GREATER_THAN;}
"<" {adjust(); return LESS_THAN;}
"==" {adjust(); return SAME_AS;}
"diamond" {adjust(); return DIAMOND;}

[a-zA-Z][a-zA-Z0-9_]*  {adjust(); yylval.sval=String(yytext); return ID;}
[0-9]+  {adjust(); yylval.ival=atoi(yytext); return NUMBER;}

<<EOF>>  {return 0;}

<COMMENT>"*/" {adjust(), BEGIN(0);}
<COMMENT>"\n" {adjust(); EM_newline();continue;}
<COMMENT>. {adjust();}

.  {adjust(); EM_error(EM_tokPos, "what kind of token is that!");}
```

A.7 main.c

```
#include <stdio.h>
#include "util.h"
#include "errormsg.h"
#include "symbol.h"
#include "abstractsyntax.h"
#include "parse.h"
#include "prettyprint.h"
#include "semantic.h"
#include "translation.h"

extern A_program abstractsyntax_root;
extern int yyparse(void);

A_program parse(string fname)
{EM_reset(fname);
 if (yyparse() == 0) /* parsing worked */
   {
     fprintf(stderr,"Parsing successful!\n");
     return abstractsyntax_root;
   }
else
  {
    fprintf(stderr,"Parsing failed\n");
    return NULL;
  }
}


int main(int argc, char *argv[])
{
  char *code;

  if (argc < 2)
    {
      fprintf(stderr,"Usage: compiler [-parse | -check] file\n");
      return 1;
    }
  else
    {
      if (strcmp(argv[1],"-parse") == 0)
{
  pretty_print_program(stdout, parse(argv[2]), 0);
}
      else
{
  if (strcmp(argv[1],"-check") == 0)
    {
      SEM_check_program(parse(argv[2]));
}
  else
    {
      code = TRANS_programcode(SEM_check_program(parse(argv[1])));
      fprintf(stdout, "%s",code);
    }
}
      return 0;
    }
```

```
}
```

A.8 parse.h

```
/* function prototype from parse.c */
A_program parse(string fname);
```

A.9 prettyprint.h

```
void pretty_print_program(FILE *out, A_program v, int indentation);
void pretty_print_expression(FILE *out, A_expression v, int indentation);
```

A.10 prettyprint.c

```
#include <stdio.h>
#include "util.h"
#include "symbol.h"
#include "abstractsyntax.h"
#include "prettyprint.h"

static void pretty_print_sig_def_list (FILE *out, A_sig_def_list v, int
indentation);
static void pretty_print_signature_definition (FILE *out,
A_signature_definition v, int indentation);
static void pretty_print_signature (FILE *out, A_signature v, int indentation);
static void pretty_print_definition (FILE *out, A_definition v, int
indentation);
static void pretty_print_variable (FILE *out, A_variable v, int indentation);
static void pretty_print_type (FILE *out, A_type v, int indentation);
static void pretty_print_variable_list (FILE *out, A_variable_list v, int
indentation);
static void pretty_print_type_list (FILE *out, A_type_list v, int indentation);
static void pretty_print_expression_list (FILE *out, A_expression_list v, int
indentation);

static void indent(FILE *out, int indentation)
{
  int i;
  for (i = 0; i <= indentation; i++)
    {
      fprintf(out, " ");
    }
}

static char string_operators[][15] = {
  "Plus", "Minus", "Multiply", "Divide", "Lessthan", "Greaterthan", "Sameas"};

static void pretty_print_operator(FILE *out, A_oper operator)
{
  fprintf(out, "%s", string_operators[operator]);
}

void pretty_print_program (FILE *out, A_program v, int indentation)
{
  indent(out, indentation);
  if (v != NULL)
    {
      fprintf(out, "Program(\n");
      pretty_print_sig_def_list(out, v->sig_def_list, indentation+1);
      fprintf(out, ")\n");
    }
}

static void pretty_print_sig_def_list (FILE *out, A_sig_def_list v, int
indentation)
{
  indent(out, indentation);
  if (v != NULL)
    {
      //fprintf(out, "\n");
      fprintf(out, "Sig and Def List(\n");
      pretty_print_signature_definition(out, v->head, indentation+1);
      fprintf(out, ",\n");
```

```
      pretty_print_sig_def_list(out, v->tail, indentation+1);
      fprintf(out, ")\n");
    }
}

static void pretty_print_signature_definition (FILE *out,
A_signature_definition v, int indentation)
{
  indent(out, indentation);
  if (v != NULL)
    {
      switch (v->kind)
{
case A_sig:
  fprintf(out, "Sig(\n");
  pretty_print_signature(out, v->u.sig, indentation+1);
  fprintf(out, ")\n");
  break;
case A_def:
  fprintf(out, "Def(\n");
  pretty_print_definition(out, v->u.def, indentation+1);
  fprintf(out, ")\n");
  break;
default:
  assert(0);
}
    }
}

static void pretty_print_signature (FILE *out, A_signature v, int indentation)
{
  indent(out, indentation);
  if (v != NULL)
    {
      fprintf(out, "Signature(%s,\n", S_name(v->name));
      pretty_print_type_list(out, v->parameter_type_list, indentation+1);
      pretty_print_type(out, v->result, indentation+1);
      fprintf(out, ")\n");
    }
}

static void pretty_print_definition (FILE *out, A_definition v, int indentation)
{
  indent(out, indentation);
  if (v != NULL)
    {
      fprintf(out, "Definition(%s,\n", S_name(v->name));
      pretty_print_variable_list(out, v->parameter_list, indentation+1);
      pretty_print_expression(out, v->body, indentation+1);
      fprintf(out, ")\n");
    }
}

static void pretty_print_variable (FILE *out, A_variable v, int indentation)
{
  indent(out, indentation);
  if (v != NULL)
    {
      fprintf(out, "Variable(%s)", S_name(v->name));
    }
}

static void pretty_print_type (FILE *out, A_type v, int indentation)
{
  indent(out, indentation);
  if (v != NULL)
    {
      switch (v->kind)
{
case A_int_Type:
  fprintf(out, "Int");
  break;
case A_diamond_Type:
  fprintf(out, "Diamond");
  break;
case A_list_Type:
  fprintf(out, "List(\n");
  pretty_print_type(out, v->u.list_type, indentation+1);
  fprintf(out, ")");
  break;
case A_tree_Type:
  fprintf(out, "Tree(\n");
  pretty_print_type(out, v->u.tree_type, indentation+1);
  fprintf(out, ")");
  break;
case A_pair_Type:
  fprintf(out, "Pair(\n");
  pretty_print_type(out, v->u.pair.left, indentation+1);
```

```
  pretty_print_type(out, v->u.pair.right, indentation+1);
  fprintf(out, ")");
  break;
default:
  assert(0);
}
    }
}

void pretty_print_expression (FILE *out, A_expression v, int indentation)
{
  indent(out, indentation);
  if (v != NULL)
    {
      switch (v->kind)
{
case A_variable_Exp:
  fprintf(out,"VariableExp(");
  pretty_print_variable(out, v->u.variable, 0);
  fprintf(out, ")");
  break;
case A_number_Exp:
  fprintf(out,"NumberExp(\n");
  fprintf(out, "%d", v->u.intt);
  fprintf(out, ")");
  break;
case A_operator_Exp:
  fprintf(out,"OperatorExp(\n");
  indent(out, indentation+1);
  pretty_print_operator(out, v->u.operator.oper);
  fprintf(out, ",\n");
  pretty_print_expression(out, v->u.operator.left, indentation+1);
  fprintf(out, ",\n");
  pretty_print_expression(out, v->u.operator.right, indentation+1);
  fprintf(out, ")");
  break;
case A_if_Exp:
  fprintf(out,"IfExp(\n");
  pretty_print_expression(out, v->u.iff.test, indentation+1);
  fprintf(out, ",\n");
  pretty_print_expression(out, v->u.iff.then, indentation+1);
  fprintf(out, ",\n");
  pretty_print_expression(out, v->u.iff.elsee, indentation+1);
  fprintf(out, ")");
  break;
case A_pair_Exp:
  fprintf(out,"PairExp(\n");
  pretty_print_expression(out, v->u.pair.left, indentation+1);
  fprintf(out, ",\n");
  pretty_print_expression(out, v->u.pair.right, indentation+1);
  fprintf(out, ")");
  break;
case A_nil_Exp:
  fprintf(out,"NilExp(\n");
  pretty_print_type(out, v->u.nilof, indentation+1);
  fprintf(out, ")");
  break;
case A_cons_Exp:
  fprintf(out,"ConsExp(\n");
  pretty_print_expression(out, v->u.cons.a, indentation+1);
  fprintf(out, ",\n");
  pretty_print_expression(out, v->u.cons.b, indentation+1);
  fprintf(out, ",\n");
  pretty_print_expression(out, v->u.cons.c, indentation+1);
  fprintf(out, ")");
  break;
case A_leaf_Exp:
  fprintf(out,"LeafExp(\n");
  pretty_print_expression(out, v->u.leaf, indentation+1);
  fprintf(out, ")\n");
  break;
case A_node_Exp:
  fprintf(out,"NodeExp(\n");
  pretty_print_expression(out, v->u.node.a, indentation+1);
  fprintf(out, ",\n");
  pretty_print_expression(out, v->u.node.b, indentation+1);
  fprintf(out, ",\n");
  pretty_print_expression(out, v->u.node.c, indentation+1);
  fprintf(out, ",\n");
  pretty_print_expression(out, v->u.node.d, indentation+1);
  fprintf(out, ",\n");
  pretty_print_expression(out, v->u.node.e, indentation+1);
  fprintf(out, ")");
  break;
case A_function_call_Exp:
  fprintf(out,"Function_CallExp(%s\n", S_name(v->u.function_call.name));
  pretty_print_expression_list(out, v->u.function_call.arguments,
indentation+1);
```

```
  indent(out, indentation);
  fprintf(out, ")");
  break;
case A_match_list_Exp:
  fprintf(out,"Match_ListExp(\n");
  pretty_print_expression(out, v->u.match_list.a, indentation+1);
  fprintf(out, ",\n");
  pretty_print_expression(out, v->u.match_list.b, indentation+1);
  fprintf(out, ",\n");
  pretty_print_variable(out, v->u.match_list.c, indentation+1);
  fprintf(out, ",\n");
  pretty_print_variable(out, v->u.match_list.d, indentation+1);
  fprintf(out, ",\n");
  pretty_print_variable(out, v->u.match_list.e, indentation+1);
  fprintf(out, ",\n");
  pretty_print_expression(out, v->u.match_list.f, indentation+1);
  fprintf(out, ")");
  break;
case A_match_pair_Exp:
  fprintf(out,"Match_PairExp(\n");
  pretty_print_expression(out, v->u.match_pair.a, indentation+1);
  fprintf(out, ",\n");
  pretty_print_variable(out, v->u.match_pair.b, indentation+1);
  fprintf(out, ",\n");
  pretty_print_variable(out, v->u.match_pair.c, indentation+1);
  fprintf(out, ",\n");
  pretty_print_expression(out, v->u.match_pair.d, indentation+1);
  fprintf(out, ")");
  break;
case A_match_tree_Exp:
  fprintf(out,"Match_TreeExp(\n");
  pretty_print_expression(out, v->u.match_tree.a, indentation+1);
  fprintf(out, ",\n");
  pretty_print_variable(out, v->u.match_tree.b, indentation+1);
  fprintf(out, ",\n");
  pretty_print_expression(out, v->u.match_tree.c, indentation+1);
  fprintf(out, ",\n");
  pretty_print_variable(out, v->u.match_tree.d, indentation+1);
  fprintf(out, ",\n");
  pretty_print_variable(out, v->u.match_tree.e, indentation+1);
  fprintf(out, ",\n");
  pretty_print_variable(out, v->u.match_tree.f, indentation+1);
  fprintf(out, ",\n");
  pretty_print_variable(out, v->u.match_tree.g, indentation+1);
  fprintf(out, ",\n");
  pretty_print_variable(out, v->u.match_tree.h, indentation+1);
  fprintf(out, ",\n");
  pretty_print_expression(out, v->u.match_tree.i, indentation+1);
  fprintf(out, ")");
  break;
default:
  assert(0);
}
    }
}

static void pretty_print_variable_list (FILE *out, A_variable_list v, int
indentation)
{
  indent(out, indentation);
  if (v != NULL)
    {
      //fprintf(out, "\n");
      fprintf(out, "Variable List(\n");
      pretty_print_variable(out, v->head, indentation+1);
      fprintf(out, ",\n");
      pretty_print_variable_list(out, v->tail, indentation+1);
      indent(out, indentation);
      fprintf(out, ")\n");
    }
}

static void pretty_print_type_list (FILE *out, A_type_list v, int indentation)
{
  indent(out, indentation);
  if (v != NULL)
    {
      //fprintf(out, "\n");
      fprintf(out, "Type List(\n");
      pretty_print_type(out, v->head, indentation+1);
      fprintf(out, ",\n");
      pretty_print_type_list(out, v->tail, indentation+1);
      indent(out, indentation);
      fprintf(out, ")\n");
    }
}

static void pretty_print_expression_list (FILE *out, A_expression_list v, int
```

```
indentation)
{
  indent(out, indentation);
  if (v != NULL)
    {
      //fprintf(out, "\n");
      fprintf(out, "Expression List(\n");
      pretty_print_expression(out, v->head, indentation+1);
      fprintf(out, ",\n");
      pretty_print_expression_list(out, v->tail, indentation+1);
      indent(out, indentation);
      fprintf(out, ")\n");
    }
}
```

A.11 semantic.h

```
/* semantic.h */

A_program SEM_check_program(A_program program);
```

A.12 semantic.c

```
/* semantic.c */

#include <stdio.h>
#include "util.h"
#include "errormsg.h"
#include "symbol.h"
#include "abstractsyntax.h"
#include "prettyprint.h"
#include "types.h"
#include "sets.h"

typedef struct SEM_environment_ *SEM_environment;

struct SEM_environment_
{
  S_table variables;
  S_table functions;
};

SEM_environment SEM_Environment(S_table variables, S_table functions);


typedef struct SEM_expression_return_ *SEM_expression_return;

struct SEM_expression_return_
{
  SET_set heap_variables;
  A_type type;
};

SEM_expression_return SEM_Expression_Return(SET_set heap_variables, A_type
type);


SEM_environment SEM_check_sig_def_list(SEM_environment environment,
A_sig_def_list l);
SEM_environment SEM_check_signature_definition(SEM_environment environment,
A_signature_definition x);
SEM_environment SEM_check_signature(SEM_environment environment, A_signature s);
void SEM_check_definition(SEM_environment environment, A_definition d);
SEM_expression_return SEM_check_expression(SEM_environment environment,
A_expression e);
A_type SEM_check_type(A_type type);
SEM_environment SEM_check_variable(SEM_environment environment, A_variable
variable, A_type formal);

A_type_list SEM_check_type_list(A_type_list list);
SEM_environment SEM_check_variable_list(SEM_environment environment,
A_variable_list parameter_list, A_type_list formals);
SEM_expression_return SEM_check_expression_list(SEM_environment environment,
A_expression_list e, A_type_list formals);

A_program SEM_check_program(A_program program)
{
  /* create empty environment */
  SEM_environment empty_environment =
```

```
(SEM_environment)checked_malloc(sizeof(*empty_environment));
  empty_environment->variables = S_empty();
  empty_environment->functions = S_empty();
  if (program != NULL)
    {
      SEM_check_sig_def_list(empty_environment, program->sig_def_list);
      fprintf(stderr, "Type Checking successful!\n");
      return program;
    }
  else
    {
      fprintf(stderr, "Type Checking successful!\n");
      return NULL;
    }
}

SEM_environment SEM_check_sig_def_list(SEM_environment environment,
A_sig_def_list l)
{

  if (l != NULL)
    {
     SEM_environment new_environment =
SEM_check_signature_definition(environment, l->head);
     return SEM_check_sig_def_list(new_environment, l->tail);
    }
  else
    {
      return environment;
    }
}

SEM_environment SEM_check_signature_definition(SEM_environment environment,
A_signature_definition x)
{
  switch (x->kind)
    {
    case A_sig:
      return SEM_check_signature(environment, x->u.sig);
      break;
    case A_def:
      SEM_check_definition(environment, x->u.def);
      return environment;
      break;
    default:
      assert(0);
    }
}

SEM_environment SEM_check_signature(SEM_environment environment, A_signature s)
{
  A_type_list a = SEM_check_type_list(s->parameter_type_list);
  A_type b = SEM_check_type(s->result);

  S_enter(environment->functions, s->name, Ty_Function(a, b));
  return environment;
}

A_type_list SEM_check_type_list(A_type_list list)
{
  if (list != NULL)
    {
      return A_Type_List(SEM_check_type(list->head),
SEM_check_type_list(list->tail));
    }
  else
    {
      return NULL;
    }
}

A_type SEM_check_type(A_type type)
{
  return type;
}

void SEM_check_definition(SEM_environment environment, A_definition d)
{
  SEM_environment new_environment;
  SEM_expression_return body;
  Ty_function x = S_look(environment->functions, d->name);
  if (x == NULL)
    {
      EM_error(d->pos, "function %s has no signature", S_name(d->name));
    }
  else
    {
      S_beginScope(environment->variables);
```

```
      new_environment = SEM_check_variable_list(environment, d->parameter_list,
x->formals);
      body = SEM_check_expression(new_environment, d->body);
      if (Ty_are_equal(body->type,
((Ty_function)(S_look(environment->functions, d->name)))->result) == FALSE)
{
  EM_error(d->body->pos, "function return type does not match
signature");
}
      S_endScope(environment->variables);
    }
}

SEM_environment SEM_check_variable_list(SEM_environment environment,
A_variable_list parameter_list, A_type_list formals)
{
  SEM_environment new_environment;
  if (parameter_list != NULL)
    {
      if (formals != NULL)
{
  new_environment = SEM_check_variable(environment,
parameter_list->head, formals->head);
  return SEM_check_variable_list(new_environment, parameter_list->tail,
formals->tail);
}
      else
{
  EM_error(parameter_list->head->pos, "function defintion contains more
parameters than signature");
}
    }
  else
    {
      if (formals != NULL)
{
  EM_error((A_pos)NULL, "function defintion contains less parameters
than signature");
}
      else
{
  return environment;
}
    }
}

SEM_environment SEM_check_variable(SEM_environment environment, A_variable
variable, A_type formal)
{

  S_enter(environment->variables, variable->name, formal);
  variable->type = formal;
  return environment;

}

SEM_expression_return SEM_check_expression(SEM_environment environment,
A_expression e)
{
  switch(e->kind)
    {
    case A_variable_Exp:
      {
A_type x = S_look(environment->variables, e->u.variable->name);
if (x == NULL)
  {
    EM_error(e->pos, "undefined variable %s",
S_name(e->u.variable->name));
  }
e->type = x;
if (Ty_is_heap(x) == TRUE)
  {
    return SEM_Expression_Return(SET_Singleton(e->u.variable->name),
x);
  }
else
  {
    return SEM_Expression_Return(SET_Empty(),x);
  }
      }
    case A_number_Exp:
      {
e->type = Ty_Int();
return SEM_Expression_Return(SET_Empty(), Ty_Int());
      }
    case A_operator_Exp:
      {
SEM_expression_return xleft = SEM_check_expression(environment,
```

```
e->u.operator.left);
SEM_expression_return xright = SEM_check_expression(environment,
e->u.operator.right);
e->type = Ty_Int();
if (xleft->type->kind != A_int_Type)
  {
    EM_error(e->u.operator.left->pos, "integer required");
  }
if (xright->type->kind != A_int_Type)
  {
     EM_error(e->u.operator.right->pos, "integer required");
  }
if (SET_Disjoint(xleft->heap_variables, xright->heap_variables) ==
FALSE)
  {
    EM_error(e->pos, "shared variable");
  }
return SEM_Expression_Return(SET_Union(xleft->heap_variables,
xright->heap_variables), Ty_Int());
      }
    case A_if_Exp:
      {
SEM_expression_return xtest = SEM_check_expression(environment,
e->u.iff.test);
SEM_expression_return xthen = SEM_check_expression(environment,
e->u.iff.then);
SEM_expression_return xelsee = SEM_check_expression(environment,
e->u.iff.elsee);
e->type = xthen->type;
if (xtest->type->kind != A_int_Type)
  {
    EM_error(e->u.iff.test->pos, "if test not of integer type");
  }
if (Ty_are_equal(xthen->type, xelsee->type) == FALSE)
  {
    EM_error(e->pos, "if, then clauses of different types ");
  }
if (SET_Disjoint(xtest->heap_variables, xthen->heap_variables) == FALSE)
  {
    EM_error(e->pos, "shared variables");
  }
if (SET_Disjoint(xtest->heap_variables, xelsee->heap_variables) ==
FALSE)
  {
    EM_error(e->pos, "shared variables");
  }
return SEM_Expression_Return(SET_Union(SET_Union(xtest->heap_variables,
xthen->heap_variables), xelsee->heap_variables), xthen->type);
      }
    case A_pair_Exp:
      {
SEM_expression_return xleft = SEM_check_expression(environment,
e->u.pair.left);
SEM_expression_return xright = SEM_check_expression(environment,
e->u.pair.right);
e->type = Ty_Pair(xleft->type,xright->type);
if (SET_Disjoint(xleft->heap_variables, xright->heap_variables) ==
FALSE)
  {
    EM_error(e->pos, "shared variables");
  }
return SEM_Expression_Return(SET_Union(xleft->heap_variables,
xright->heap_variables), Ty_Pair(xleft->type,xright->type));
      }
    case A_nil_Exp:
      {
e->type = Ty_List(SEM_check_type(e->u.nilof));
return SEM_Expression_Return(SET_Empty(),
Ty_List(SEM_check_type(e->u.nilof)));
      }
    case A_cons_Exp:
      {
SEM_expression_return xa = SEM_check_expression(environment,
e->u.cons.a);
SEM_expression_return xb = SEM_check_expression(environment,
e->u.cons.b);
SEM_expression_return xc = SEM_check_expression(environment,
e->u.cons.c);
e->type = xc->type;
if (xa->type->kind != A_diamond_Type)
  {
    EM_error(e->u.cons.a->pos, "not of type diamond");
  }
if (Ty_are_equal(xc->type, Ty_List(xb->type)) == FALSE)
  {
    Ty_type_print(xc->type);
    fprintf(stderr, "\n");
    Ty_type_print(xb->type);
```

```
    fprintf(stderr, "\n");
    EM_error(e->pos, "tail not of list(head) type");
  }
if (SET_Disjoint(xa->heap_variables, xb->heap_variables) == FALSE)
  {
    EM_error(e->pos, "shared variables");
  }
if (SET_Disjoint(xb->heap_variables, xc->heap_variables) == FALSE)
  {
    EM_error(e->pos, "shared variables");
  }
if (SET_Disjoint(xa->heap_variables, xc->heap_variables) == FALSE)
  {
    EM_error(e->pos, "shared variables");
  }
return SEM_Expression_Return(SET_Union(SET_Union(xa->heap_variables,
xb->heap_variables),xc->heap_variables), xc->type);
      }
    case A_leaf_Exp:
      {
SEM_expression_return xleaf = SEM_check_expression(environment,
e->u.leaf);
e->type = Ty_Tree(xleaf->type);
return SEM_Expression_Return(xleaf->heap_variables,
Ty_Tree(xleaf->type));
      }
    case A_node_Exp:
      {
SEM_expression_return xa = SEM_check_expression(environment,
e->u.node.a);
SEM_expression_return xb = SEM_check_expression(environment,
e->u.node.b);
SEM_expression_return xc = SEM_check_expression(environment,
e->u.node.c);
SEM_expression_return xd = SEM_check_expression(environment,
e->u.node.d);
SEM_expression_return xe = SEM_check_expression(environment,
e->u.node.e);
e->type = xd->type;
if (xa->type->kind != A_diamond_Type)
  {
     EM_error(e->u.node.a->pos, "not of type diamond");
  }
if (xb->type->kind != A_diamond_Type)
  {
    EM_error(e->u.node.b->pos, "not of type diamond");
  }
if (Ty_are_equal(xd->type, Ty_Tree(xc->type)) == FALSE)
  {
     EM_error(e->pos, "branch not of tree(parent) type");
  }
if (Ty_are_equal(xe->type, Ty_Tree(xc->type)) == FALSE)
  {
    EM_error(e->pos, "branch not of tree(parent) type");
  }
if (SET_Disjoint(xa->heap_variables, xb->heap_variables) == FALSE)
  {
    EM_error(e->pos, "shared variables");
  }
if (SET_Disjoint(xa->heap_variables, xc->heap_variables) == FALSE)
  {
    EM_error(e->pos, "shared variables");
  }
if (SET_Disjoint(xa->heap_variables, xd->heap_variables) == FALSE)
  {
    EM_error(e->pos, "shared variables");
  }
if (SET_Disjoint(xa->heap_variables, xe->heap_variables) == FALSE)
  {
    EM_error(e->pos, "shared variables");
  }
if (SET_Disjoint(xb->heap_variables, xc->heap_variables) == FALSE)
  {
    EM_error(e->pos, "shared variables");
  }
if (SET_Disjoint(xb->heap_variables, xd->heap_variables) == FALSE)
  {
    EM_error(e->pos, "shared variables");
  }
if (SET_Disjoint(xb->heap_variables, xe->heap_variables) == FALSE)
  {
    EM_error(e->pos, "shared variables");
  }
if (SET_Disjoint(xc->heap_variables, xd->heap_variables) == FALSE)
  {
    EM_error(e->pos, "shared variables");
  }
if (SET_Disjoint(xc->heap_variables, xe->heap_variables) == FALSE)
```

```
  {
    EM_error(e->pos, "shared variables");
  }
if (SET_Disjoint(xd->heap_variables, xe->heap_variables) == FALSE)
  {
    EM_error(e->pos, "shared variables");
  }
return
SEM_Expression_Return(SET_Union(SET_Union(SET_Union(SET_Union(xa->heap_variables
, xb->heap_variables), xc->heap_variables), xd->heap_variables),
xe->heap_variables), xd->type);
      }
    case A_function_call_Exp:
      {
/* to do */
SEM_expression_return xexp_list;
Ty_function x = S_look(environment->functions, e->u.function_call.name);
if (x == NULL)
  {
    EM_error(e->pos, "undefined function called %s",
S_name(e->u.function_call.name));
  }
e->type = x->result;
xexp_list = SEM_check_expression_list(environment,
e->u.function_call.arguments, x->formals);
return SEM_Expression_Return(xexp_list->heap_variables, x->result);
      }
    case A_match_list_Exp:
      {
SEM_expression_return xa;
SEM_expression_return xb;
SEM_expression_return xf;
xa = SEM_check_expression(environment, e->u.match_list.a);
xb = SEM_check_expression(environment, e->u.match_list.b);
e->type = xb->type;
if (xa->type->kind != A_list_Type)
  {
    EM_error(e->u.match_list.a->pos, "match list expression is not of
list type");
  }
S_beginScope(environment->variables);
    S_enter(environment->variables, e->u.match_list.c->name, Ty_Diamond());
S_enter(environment->variables, e->u.match_list.d->name,
xa->type->u.list_type);
S_enter(environment->variables, e->u.match_list.e->name, xa->type);
xf = SEM_check_expression(environment, e->u.match_list.f);
S_endScope(environment->variables);
if (Ty_are_equal(xb->type, xf->type) == FALSE)
  {
    EM_error(e->pos, "match return clauses of different types ");
  }
xf->heap_variables = SET_Remove(xf->heap_variables,
e->u.match_list.c->name);
xf->heap_variables = SET_Remove(xf->heap_variables,
e->u.match_list.d->name);
xf->heap_variables = SET_Remove(xf->heap_variables,
e->u.match_list.e->name);
if (SET_Disjoint(xa->heap_variables, xb->heap_variables) == FALSE)
  {
    EM_error(e->pos, "shared variables");
  }
if (SET_Disjoint(xa->heap_variables, xf->heap_variables) == FALSE)
  {
    EM_error(e->pos, "shared variables");
  }
return SEM_Expression_Return(SET_Union(SET_Union(xa->heap_variables,
xb->heap_variables), xf->heap_variables), xb->type);
      }
    case A_match_pair_Exp:
      {
SEM_expression_return xa;
SEM_expression_return xd;
xa = SEM_check_expression(environment, e->u.match_pair.a);
if (xa->type->kind != A_pair_Type)
  {
    Ty_type_print(xa->type);
    fprintf(stderr, "\n");
    pretty_print_expression(stderr, e->u.match_pair.a, 0);
    EM_error(e->u.match_pair.a->pos, "match pair expression is not of
pair type");
  }
S_beginScope(environment->variables);
S_enter(environment->variables, e->u.match_pair.b->name,
xa->type->u.pair.left);
S_enter(environment->variables, e->u.match_pair.c->name,
xa->type->u.pair.right);
xd = SEM_check_expression(environment, e->u.match_pair.d);
e->type = xd->type;
```

```
S_endScope(environment->variables);
xd->heap_variables = SET_Remove(xd->heap_variables,
e->u.match_pair.b->name);
xd->heap_variables = SET_Remove(xd->heap_variables,
e->u.match_pair.c->name);
if (SET_Disjoint(xa->heap_variables, xd->heap_variables) == FALSE)
  {
    EM_error(e->pos, "shared variables");
  }
return SEM_Expression_Return(SET_Union(xa->heap_variables,
xd->heap_variables), xd->type);
      }
    case A_match_tree_Exp:
      {
SEM_expression_return xa;
SEM_expression_return xc;
SEM_expression_return xi;
xa = SEM_check_expression(environment, e->u.match_tree.a);
if (xa->type->kind != A_tree_Type)
  {
    EM_error(e->u.match_tree.a->pos, "match tree expression is not of
tree type");
  }
S_beginScope(environment->variables);
S_enter(environment->variables, e->u.match_tree.b->name,
xa->type->u.tree_type);
xc = SEM_check_expression(environment, e->u.match_tree.c);
e->type = xc->type;
S_endScope(environment->variables);

S_beginScope(environment->variables);
S_enter(environment->variables, e->u.match_tree.d->name, Ty_Diamond());
S_enter(environment->variables, e->u.match_tree.e->name, Ty_Diamond());
S_enter(environment->variables, e->u.match_tree.b->name,
xa->type->u.tree_type);
S_enter(environment->variables, e->u.match_tree.g->name, xa->type);
S_enter(environment->variables, e->u.match_tree.h->name, xa->type);
xi = SEM_check_expression(environment, e->u.match_tree.i);
S_endScope(environment->variables);
if (Ty_are_equal(xc->type, xi->type) == FALSE)
  {
    EM_error(e->pos, "match return clauses of different types");
  }
xc->heap_variables = SET_Remove(xc->heap_variables,
e->u.match_tree.b->name);
xi->heap_variables = SET_Remove(xi->heap_variables,
e->u.match_tree.d->name);
xi->heap_variables = SET_Remove(xi->heap_variables,
e->u.match_tree.e->name);
xi->heap_variables = SET_Remove(xi->heap_variables,
e->u.match_tree.b->name);
xi->heap_variables = SET_Remove(xi->heap_variables,
e->u.match_tree.g->name);
xi->heap_variables = SET_Remove(xi->heap_variables,
e->u.match_tree.h->name);
if (SET_Disjoint(xa->heap_variables, xc->heap_variables) == FALSE)
  {
    EM_error(e->pos, "shared variables");
  }
if (SET_Disjoint(xa->heap_variables, xi->heap_variables) == FALSE)
  {
    EM_error(e->pos, "shared variables");
  }
return SEM_Expression_Return(SET_Union(SET_Union(xa->heap_variables,
xc->heap_variables), xi->heap_variables), xc->type);
      }
    default:
      assert(0);
    }
}

SEM_expression_return SEM_check_expression_list(SEM_environment environment,
A_expression_list e, A_type_list formals)
{
  SEM_expression_return exp;
  SEM_expression_return exp_list;
  if (e != NULL)
    {
      if (formals != NULL)
{
  exp = SEM_check_expression(environment, e->head);
  if (Ty_are_equal(exp->type, formals->head) == FALSE)
    {
      EM_error(e->head->pos, "argument type does not match signature");
    }
  exp_list = SEM_check_expression_list(environment, e->tail,
formals->tail);
  if (SET_Disjoint(exp->heap_variables, exp_list->heap_variables) ==
```

```
FALSE)
    {
      EM_error(e->head->pos, "shared variables");
    }
  return SEM_Expression_Return(SET_Union(exp->heap_variables,
exp_list->heap_variables), (A_type)NULL);
}
      else
{
  EM_error(e->head->pos, "function called with too many arguments");
}
    }
  else
    {
      if (formals != NULL)
{
  EM_error((A_pos)NULL, "function called with too few arguments");
}
      else
{
return SEM_Expression_Return(SET_Empty(), (A_type)NULL);
}
    }
}

SEM_expression_return SEM_Expression_Return(SET_set heap_variables, A_type type)
{
  SEM_expression_return p = (SEM_expression_return)checked_malloc(sizeof(*p));
  p->heap_variables = heap_variables;
  p->type = type;
  return p;
}
```

A.13 sets.h

```
/* sets.h */

typedef struct SET_set_ *SET_set;

struct SET_set_
{
  S_symbol data;
  SET_set next;
};

SET_set SET_Empty(void);

SET_set SET_Singleton(S_symbol element);

SET_set SET_Add(SET_set set, S_symbol element);

SET_set SET_Union(SET_set set1, SET_set set2);

int SET_IsMember(SET_set set, S_symbol element);

SET_set SET_Remove(SET_set set, S_symbol element);

int SET_Disjoint(SET_set set1, SET_set set2);

SET_set SET_Intersection(SET_set set1, SET_set set2);
```

A.14 sets.c

```
/* sets.c */

#include <stdio.h>
#include "util.h"
#include "errormsg.h"
#include "symbol.h"
#include "sets.h"


SET_set SET_Empty(void)
{
  return NULL;
}

SET_set SET_Singleton(S_symbol element)
{
```

```
  SET_set s = (SET_set)checked_malloc(sizeof(*s));
  s->data = element;
  s->next = NULL;
  return s;
}

SET_set SET_Add(SET_set set, S_symbol element)
{
  if (!SET_IsMember(set,element))
    {
      SET_set s = (SET_set)checked_malloc(sizeof(*s));
      s->data = element;
      s-> next = set;
      return s;
    }
  else
    {
      return set;
    }
}

int SET_Disjoint(SET_set set1, SET_set set2)
{
  if (set1 != NULL)
    {
      if (SET_IsMember(set2,set1->data))
{
  return FALSE;
}
      else
{
  return SET_Disjoint((set1->next),set2);
}
    }
  else
    {
      return TRUE;
    }
}

int SET_IsMember(SET_set set, S_symbol element)
{
  if (set != NULL)
    {
      if (element == set->data)
{
  return TRUE;
}
      else
{
  return SET_IsMember(set->next, element);
}
    }
  else
    {
      return FALSE;
    }
}

SET_set SET_Remove(SET_set set, S_symbol element)
{
  SET_set ptr = NULL;
  SET_set prevptr = NULL;
  for (ptr = set; ptr != NULL; ptr = ptr->next)
    {
      if (element == ptr->data)
{
  break;
}
      prevptr = ptr;
    }
  if (ptr == NULL)
    {
      return set;
    }
  if (prevptr != NULL)
    {
      prevptr->next = ptr->next;
    }
  else
    {
      set = ptr->next;
    }
  return set;
}

SET_set SET_Union(SET_set set1, SET_set set2)
{
```

```
  if (set1 != NULL)
    {
      return SET_Union(set1->next , SET_Add(set2, set1->data));
    }
  else
    {
      return set2;
    }
}

SET_set SET_Intersection(SET_set set1, SET_set set2)
{
  if (set1 != NULL)
    {
      if (SET_IsMember(set2, set1->data) == TRUE)
{
  return SET_Union(SET_Singleton(set1->data),
SET_Intersection(set1->next, set2));
}
      else
{
  return SET_Intersection(set1->next, set2);
}
    }
  else
    {
      return SET_Empty();
    }
}
```

A.15 symbol.h

```
/*
 * symbol.h - Symbols and symbol-tables
 *
 */

typedef struct S_symbol_ *S_symbol;

/* Make a unique symbol from a given string.
 *  Different calls to S_Symbol("foo") will yield the same S_symbol
 *  value, even if the "foo" strings are at different locations. */
S_symbol S_Symbol(string);

/* Extract the underlying string from a symbol */
string S_name(S_symbol);

/* S_table is a mapping from S_symbol->any, where "any" is represented
 *     here by void*  */
typedef struct TAB_table_ *S_table;

/* Make a new table */
S_table S_empty(void);

/* Enter a binding "sym->value" into "t", shadowing but not deleting
 *    any previous binding of "sym". */
void S_enter(S_table t, S_symbol sym, void *value);

/* Look up the most recent binding of "sym" in "t", or return NULL
 *    if sym is unbound. */
void *S_look(S_table t, S_symbol sym);

/* Start a new "scope" in "t".  Scopes are nested. */
void S_beginScope(S_table t);

/* Remove any bindings entered since the current scope began,
   and end the current scope. */
void S_endScope(S_table t);
```

A.16 symbol.c

```
#include <stdio.h>
#include <string.h>
#include "util.h"
#include "symbol.h"
#include "table.h"
```

```
struct S_symbol_ {string name; S_symbol next;};

static S_symbol mksymbol(string name, S_symbol next)
{S_symbol s=checked_malloc(sizeof(*s));
 s->name=name; s->next=next;
 return s;
}

#define SIZE 109  /* should be prime */

static S_symbol hashtable[SIZE];

static unsigned int hash(char *s0)
{unsigned int h=0; char *s;
 for(s=s0; *s; s++)
       h = h*65599 + *s;
 return h;
}

static int streq(string a, string b)
{
 return !strcmp(a,b);
}

S_symbol S_Symbol(string name)
{int index= hash(name) % SIZE;
 S_symbol syms = hashtable[index], sym;
 for(sym=syms; sym; sym=sym->next)
   if (streq(sym->name,name)) return sym;
 sym = mksymbol(name,syms);
 hashtable[index]=sym;
 return sym;
}

string S_name(S_symbol sym)
{
 return sym->name;
}

S_table S_empty(void)
{
 return TAB_empty();
}

void S_enter(S_table t, S_symbol sym, void *value) {
  TAB_enter(t,sym,value);
}

void *S_look(S_table t, S_symbol sym) {
  return TAB_look(t,sym);
}

static struct S_symbol_ marksym = {"<mark>",0};

void S_beginScope(S_table t)
{ S_enter(t,&marksym,NULL);
}

void S_endScope(S_table t)
{S_symbol s;
  do s=TAB_pop(t);
  while (s != &marksym);
}

void S_dump(S_table t, void (*show)(S_symbol sym, void *binding)) {
  TAB_dump(t, (void (*)(void *, void *)) show);
}
```

A.17 table.h

```
/*
 * table.h - generic hash table
 *
 * No algorithm should use these functions directly, because
 *  programming with void* is too error-prone.  Instead,
 *  each module should make "wrapper" functions that take
 *  well-typed arguments and call the TAB_ functions.
 */

typedef struct TAB_table_ *TAB_table;

/* Make a new table mapping "keys" to "values". */
```

```
TAB_table TAB_empty(void);

/* Enter the mapping "key"->"value" into table "t",
 *    shadowing but not destroying any previous binding for "key". */
void TAB_enter(TAB_table t, void *key, void *value);

/* Look up the most recent binding for "key" in table "t" */
void *TAB_look(TAB_table t, void *key);

/* Pop the most recent binding and return its key.
 * This may expose another binding for the same key, if there was one. */
void *TAB_pop(TAB_table t);


/* Call "show" on every "key"->"value" pair in the table,
 *  including shadowed bindings, in order from the most
 *  recent binding of any key to the oldest binding in the table */
void TAB_dump(TAB_table t, void (*show)(void *key, void *value));
```

A.18 table.c

```
/*
 * table.c - Functions to manipulate generic tables.
 * Copyright (c) 1997 Andrew W. Appel.
 */

#include <stdio.h>
#include "util.h"
#include "table.h"

#define TABSIZE 127

typedef struct binder_ *binder;
struct binder_ {void *key; void *value; binder next; void *prevtop;};
struct TAB_table_ {
  binder table[TABSIZE];
  void *top;
};


static binder Binder(void *key, void *value, binder next, void *prevtop)
{binder b = checked_malloc(sizeof(*b));
 b->key = key; b->value=value; b->next=next; b->prevtop = prevtop;
 return b;
}

TAB_table TAB_empty(void)
{
 TAB_table t = checked_malloc(sizeof(*t));
 int i;
 t->top = NULL;
 for (i = 0; i < TABSIZE; i++)
   t->table[i] = NULL;
 return t;
}

/* The cast from pointer to integer in the expression
 *   ((unsigned)key) % TABSIZE
 * may lead to a warning message.  However, the code is safe,
 * and will still operate correctly.  This line is just hashing
 * a pointer value into an integer value, and no matter how the
 * conversion is done, as long as it is done consistently, a
 * reasonable and repeatable index into the table will result.
 */

void TAB_enter(TAB_table t, void *key, void *value)
{int index;
 assert(t && key);
 index = ((unsigned)key) % TABSIZE;
 t->table[index] = Binder(key, value,t->table[index], t->top);
 t->top = key;
}

void *TAB_look(TAB_table t, void *key)
{int index;
 binder b;
 assert(t && key);
 index=((unsigned)key) % TABSIZE;
 for(b=t->table[index]; b; b=b->next)
   if (b->key==key) return b->value;
 return NULL;
```

```
}

void *TAB_pop(TAB_table t) {
  void *k; binder b; int index;
  assert (t);
  k = t->top;
  assert (k);
  index = ((unsigned)k) % TABSIZE;
  b = t->table[index];
  assert(b);
  t->table[index] = b->next;
  t->top=b->prevtop;
  return b->key;
}

void TAB_dump(TAB_table t, void (*show)(void *key, void *value)) {
  void *k = t->top;
  int index = ((unsigned)k) % TABSIZE;
  binder b = t->table[index];
  if (b==NULL) return;
  t->table[index]=b->next;
  t->top=b->prevtop;
  show(b->key,b->value);
  TAB_dump(t,show);
  assert(t->top == b->prevtop && t->table[index]==b->next);
  t->top=k;
  t->table[index]=b;
}
```

A.19 translation.h

```
char *TRANS_programcode(A_program program);
```

A.20 translation.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "util.h"
#include "errormsg.h"
#include "symbol.h"
#include "abstractsyntax.h"
#include "types.h"
#include "sets.h"
#include "variables.h"

/* these are concatanated to make a program */
char *types;
char *prototypes;
char *functions;

/* used to generate fresh variable names in function declarations */
int counter;

/* these declarations are put at the start of a function */
char *declarations;

/* symbol table containing the C string representing a variable */
S_table variables;

/* global, list of types for which code has been produced */
A_type_list produced_already = NULL;


/* prototypes */
A_type_list TRANS_add_to_list(A_type type, A_type_list list);
int TRANS_types_are_equal(A_type a, A_type b);
int TRANS_is_in_list(A_type type, A_type_list list);
char *TRANS_typename(A_type type);
char *TRANS_typecode(A_type type);
char *TRANS_recursive_typecode(A_type type);
char *TRANS_operator(A_oper operator);
char *TRANS_expression_listcode(A_expression_list expression_list);
char * TRANS_expressioncode(A_expression expression);
char *TRANS_variablecode(A_variable variable);
char *TRANS_variable_listcode(A_variable_list variable_list);
void TRANS_definitioncode(A_definition definition);
char *TRANS_type_listcode(A_type_list type_list);
void TRANS_signaturecode(A_signature signature);
```

```
void TRANS_signature_definitioncode(A_signature_definition
signature_definition);
void TRANS_sig_def_listcode(A_sig_def_list sig_def_list);
char *concat(char *a, char *b);
char *concat_with_free(char *a, char *b);



char *concat(char *a, char *b)
{
  char *temp;
  temp = calloc(strlen(a) + strlen(b) + 1, sizeof(char));
  strcpy(temp, a);
  strcat(temp, b);
  return temp;
}

char *concat_with_free(char *a, char *b)
{
  char *temp;
  temp = calloc(strlen(a) + strlen(b) + 1, sizeof(char));
  strcpy(temp, a);
  strcat(temp, b);
  free(a);
  free(b);
  return temp;
}


A_type_list TRANS_add_to_list(A_type type, A_type_list list)
{
  A_type_list l = (A_type_list)checked_malloc(sizeof(*l));
  l->head = type;
  l->tail = list;
  return l;
}

int TRANS_types_are_equal(A_type a, A_type b)
{
  switch (a->kind)
    {
    case A_int_Type:
      if (b->kind == A_int_Type)
{
  return TRUE;
}
      else
{
  return FALSE;
}
    case A_diamond_Type:
      if (b->kind == A_diamond_Type)
{
  return TRUE;
}
      else
{
  return FALSE;
}
    case A_list_Type:
      if (b->kind == A_list_Type)
{
  return (TRANS_types_are_equal(a->u.list_type, b->u.list_type));
}
      else
{
  return FALSE;
}
    case A_tree_Type:
      if (b->kind == A_tree_Type)
{
  return (TRANS_types_are_equal(a->u.tree_type, b->u.tree_type));
}
      else
{
  return FALSE;
}
    case A_pair_Type:
      if (b->kind == A_pair_Type)
        {
          return (TRANS_types_are_equal(a->u.pair.left, b->u.pair.left) &&
TRANS_types_are_equal(a->u.pair.right, b->u.pair.right));
        }
      else
        {
          return FALSE;
        }
```

```
    default:
      assert(0);
    }
}

int TRANS_is_in_list(A_type type, A_type_list list)
{

  /* returns true if type is in list */
  /* return false otherwise */

  if (list == NULL)
    {
      return FALSE;
    }
  else
    {
      if (TRANS_types_are_equal(type, list->head) == TRUE)
{
  return TRUE;
}
      else
{
  return TRANS_is_in_list(type, list->tail);
}
    }
}



char *TRANS_typename(A_type type)
{

  /* simply produces a types C name */

  char *name;
  char *temp;
  char *temp2;

  switch (type->kind)
    {
    case A_int_Type:
      {
name = calloc(2, sizeof(char));
strcpy(name, "i");
return name;
      }
    case A_diamond_Type:
      {
name = calloc(2, sizeof(char));
strcpy(name, "d");
return name;
      }
    case A_pair_Type:
      {
temp = TRANS_typename(type->u.pair.left);
temp2 = TRANS_typename(type->u.pair.right);
name = calloc(strlen(temp) + strlen(temp2) + 2, sizeof(char));
strcpy(name, "p");
strcat(name, temp);
strcat(name, temp2);
free(temp);
free(temp2);
return name;
      }
    case A_list_Type:
      {
temp = TRANS_typename(type->u.list_type);
name = calloc(strlen(temp)+2, sizeof(char));
strcpy(name, "l");
strcat(name, temp);
free(temp);
return name;
      }
    case A_tree_Type:
      {
temp = TRANS_typename(type->u.tree_type);
name = calloc(strlen(temp)+2, sizeof(char));
strcpy(name, "t");
strcat(name, temp);
free(temp);
return name;
      }
    default :
      assert(0);
    }
}
```

```
char *TRANS_typecode(A_type type)
{

  /* simply produces the C code for a type */
  /* no checking to see if has been produced already is done */

  char *code;
  char *temp = TRANS_typename(type);
  char *temp2;
  char *temp3;

  switch (type->kind)
    {
    case A_int_Type:
      {
code = calloc(strlen(temp) + 15, sizeof(char));
strcpy(code, "typedef int ");
strcat(code, temp);
strcat(code, ";\n");
free(temp);
return code;
      }
    case A_diamond_Type:
      {
code = calloc(strlen(temp) + 17, sizeof(char));
strcpy(code, "typedef void *");
strcat(code, temp);
strcat(code, ";\n");
free(temp);
return code;
      }
    case A_pair_Type:
      {
temp2 = TRANS_typename(type->u.pair.left);
temp3 = TRANS_typename(type->u.pair.right);
code = calloc(4 * (strlen(temp)) + (2 * strlen(temp2)) + (2 *
strlen(temp3)) + 105, sizeof(char));
strcpy(code, "typedef struct{\n");
strcat(code, temp2);
strcat(code, " fst;\n");
strcat(code, temp3);
strcat(code, " snd;\n} ");
strcat(code, temp);
strcat(code, ";\n\n");
strcat(code, temp);
strcat(code, " ");
strcat(code, temp);
strcat(code, "_pair(");
strcat(code, temp2);
strcat(code, " fst, ");
strcat(code, temp3);
strcat(code, " snd)\n{\n");
strcat(code, temp);
strcat(code, " res;\nres.fst = fst;\nres.snd = snd;\nreturn res;\n}\n");
free(temp);
free(temp2);
free(temp3);
return code;
      }
    case A_list_Type:
      {
temp2 = TRANS_typename(type->u.list_type);
code = calloc((16 * strlen(temp)) + (3 * strlen(temp2)) + 410,
sizeof(char));
strcpy(code, "typedef struct{\nkind_t kind;\n");
strcat(code, temp2);
strcat(code, " hd;\nvoid *tl;\n} ");
strcat(code, temp);
strcat(code, ";\n\n");
strcat(code, temp);
strcat(code, " ");
strcat(code, temp);
strcat(code, "_nil(void)\n{\n");
strcat(code, temp);
strcat(code, " res;\nres.kind = NIL;\nreturn res;\n}\n\n");
strcat(code, temp);
strcat(code, " ");
strcat(code, temp);
strcat(code, "_cons(");
strcat(code, TRANS_typename(A_Diamond_Type((A_pos)NULL)));
strcat(code, " d, ");
strcat(code, temp2);
strcat(code, " hd, ");
strcat(code, temp);
strcat(code, " tl)\n{\n");
strcat(code, temp);
strcat(code, " res;\nres.kind = CONS;\nres.hd = hd;\n*(");
strcat(code, temp);
```

```
strcat(code, " *)d = tl;\nres.tl = d;\nreturn res;\n}\n\ntypedef
struct{\nkind_t kind;\n");
strcat(code, temp2);
strcat(code, " hd;\n");
strcat(code, temp);
strcat(code, " tl;\n");
strcat(code, TRANS_typename(A_Diamond_Type((A_pos)NULL)));
strcat(code, " d;\n}");
strcat(code, temp);
strcat(code, "_destr_t;\n\n");
strcat(code, temp);
strcat(code, "_destr_t ");
strcat(code, temp);
strcat(code, "_destr(");
strcat(code, temp);
strcat(code, " l)\n{\n");
strcat(code, temp);
strcat(code, "_destr_t res;\nres.kind = l.kind;\nif(res.kind ==
CONS)\n{\nres.hd = l.hd;\nres.d = (void *)l.tl;\nres.tl = *(");
strcat(code, temp);
strcat(code, " *)l.tl;\n}\nreturn res;\n}\n");
free(temp);
free(temp2);
return code;
      }
    case A_tree_Type:
      {
temp2 = TRANS_typename(type->u.tree_type);
code = calloc(20 * (strlen(temp)) + (4 * strlen(temp2)) + 585,
sizeof(char));
strcpy(code, "typedef struct{\nkind_t kind;\n");
strcat(code, temp2);
strcat(code, " label;\nvoid *left;\nvoid *right;\n} ");
strcat(code, temp);
strcat(code, ";\n\n");
strcat(code, temp);
strcat(code, " ");
strcat(code, temp);
strcat(code, "_leaf(");
strcat(code, temp2);
strcat(code, " label)\n{\n");
strcat(code, temp);
strcat(code, " res;\nres.kind = LEAF;\nres.label = label;\nreturn
res;\n}\n\n");
strcat(code, temp);
strcat(code, " ");
strcat(code, temp);
strcat(code, "_node(");
strcat(code, TRANS_typename(A_Diamond_Type((A_pos)NULL)));
strcat(code, " d1, ");
strcat(code, TRANS_typename(A_Diamond_Type((A_pos)NULL)));
strcat(code, " d2, ");
strcat(code, temp2);
strcat(code, " label, ");
strcat(code, temp);
strcat(code, " left, ");
strcat(code, temp);
strcat(code, " right)\n{\n");
strcat(code, temp);
strcat(code, " res;\nres.kind = NODE;\nres.label = label;\n*(");
strcat(code, temp);
strcat(code, " *)d1 = left;\n*(");
strcat(code, temp);
strcat(code, " *)d2 = right;\nres.left = d1;\nres.right = d2;\nreturn
res;\n}\n\ntypedef struct{\nkind_t kind;\n");
strcat(code, temp2);
strcat(code, " label;\n");
strcat(code, TRANS_typename(A_Diamond_Type((A_pos)NULL)));
strcat(code, " d1, d2;\n");
strcat(code, temp);
strcat(code, " left;\n");
strcat(code, temp);
strcat(code, " right;\n}");
strcat(code, temp);
strcat(code, "_destr_t;\n\n");
strcat(code, temp);
strcat(code, "_destr_t ");
strcat(code, temp);
strcat(code, "_destr(");
strcat(code, temp);
strcat(code, " t)\n{\n");
strcat(code, temp);
strcat(code, "_destr_t res;\nres.label = t.label;\nif((res.kind =
t.kind) == NODE)\n{\nres.d1 = (");
strcat(code, TRANS_typename(A_Diamond_Type((A_pos)NULL)));
strcat(code, ")t.left;\nres.d2 = (");
strcat(code, TRANS_typename(A_Diamond_Type((A_pos)NULL)));
strcat(code, ")t.right;\nres.left = *(");
```

```
strcat(code, temp);
strcat(code, " *)t.left;\nres.right = *(");
strcat(code, temp);
strcat(code, " *)t.right;\n}\nreturn res;\n}\n");
free(temp);
free(temp2);
return code;
      }
    default :
      assert(0);
    }
}

char *TRANS_recursive_typecode(A_type type)
{

  /* recures through a type */
  /* produces code for the type and all sub types */
  /* checks that code is not duplicated */

  char *code;
  char *temp;
  char *temp2;
  char *temp3;

  if (TRANS_is_in_list(type, produced_already) == TRUE)
    {
      code = calloc(1, sizeof(char));
      strcpy(code, "");
      return code;
    }
  else
    {
      produced_already = TRANS_add_to_list(type, produced_already);
      switch (type->kind)
{
case A_int_Type:
  {
    return TRANS_typecode(type);
  }
case A_diamond_Type:
  {
    return TRANS_typecode(type);
  }
case A_pair_Type:
  {
    temp = TRANS_typecode(type);
    temp2 = TRANS_recursive_typecode(type->u.pair.left);
    temp3 = TRANS_recursive_typecode(type->u.pair.right);
    code = calloc(strlen(temp) + strlen(temp2) + strlen(temp3) + 4,
sizeof(char));
    strcpy(code, temp2);
    strcat(code, "\n");
    strcat(code, temp3);
    strcat(code, "\n");
    strcat(code, temp);
    strcat(code, "\n");
    free(temp);
    free(temp2);
    free(temp3);
    return code;
  }
case A_list_Type:
  {
    temp = TRANS_typecode(type);
    temp2 = TRANS_recursive_typecode(type->u.list_type);
    temp3 = TRANS_recursive_typecode(A_Diamond_Type((A_pos)NULL));
    code = calloc(strlen(temp) + strlen(temp2) + strlen(temp3) + 4,
sizeof(char));
    strcpy(code, temp3);
    strcat(code, "\n");
    strcat(code, temp2);
    strcat(code, "\n");
    strcat(code, temp);
    strcat(code, "\n");
    free(temp);
    free(temp2);
    free(temp3);
    return code;
  }
case A_tree_Type:
  {
    temp = TRANS_typecode(type);
    temp2 = TRANS_recursive_typecode(type->u.tree_type);
    temp3 = TRANS_recursive_typecode(A_Diamond_Type((A_pos)NULL));
    code = calloc(strlen(temp) + strlen(temp2) + strlen(temp3) + 4,
sizeof(char));
    strcpy(code, temp3);
```

```
    strcat(code, "\n");
    strcpy(code, temp2);
    strcat(code, "\n");
    strcat(code, temp);
    strcat(code, "\n");
    free(temp);
    free(temp2);
    free(temp3);
    return code;
  }
default:
  assert(0);
}
    }
}

char *TRANS_operator(A_oper operator)
{
  char *code;
  switch (operator)
    {
    case A_plusOp:
      code = calloc(2, sizeof(char));
      strcpy(code, "+");
      return code;
      break;
    case A_minusOp:
      code = calloc(2, sizeof(char));
      strcpy(code, "-");
      return code;
      break;
    case A_multiplyOp:
      code = calloc(2, sizeof(char));
      strcpy(code, "*");
      return code;
      break;
    case A_divideOp:
      code = calloc(2, sizeof(char));
      strcpy(code, "/");
      return code;
      break;
    case A_lessOp:
      code = calloc(2, sizeof(char));
      strcpy(code, "<");
      return code;
      break;
    case A_greaterOp:
      code = calloc(2, sizeof(char));
      strcpy(code, ">");
      return code;
      break;
    case A_sameOp:
      code = calloc(3, sizeof(char));
      strcpy(code, "==");
      return code;
      break;
    default:
      assert(0);
    }
}

char *TRANS_expression_listcode(A_expression_list expression_list)
{
  char *temp;
  char *temp2;
  char *code;

  if (expression_list != NULL)
    {
      temp = TRANS_expressioncode(expression_list->head);
      temp2 = TRANS_expression_listcode(expression_list->tail);
      code = calloc(strlen(temp) + strlen(temp2) + 3, sizeof(char));
      strcpy(code, temp);
      if (strlen(temp2) != 0)
{
  strcat(code, ", ");
  strcat(code, temp2);
}
      free(temp);
      free(temp2);
      return code;
    }
  else
    {
      code = calloc(1, sizeof(char));
      strcpy(code, "");
      return code;
    }
```

```
}

char *TRANS_expressioncode(A_expression expression)
{

  char *code;
  char *temp;
  char *temp2;
  char *temp3;
  char *temp4;
  char *temp5;
  char *temp6;
  char *temp7;
  char *temp8;
  char *temp9;
  char *temp10;
  char *temp11;

  switch (expression->kind)
    {
    case A_variable_Exp:
      code = calloc(strlen(S_look(variables, expression->u.variable->name)) +
1, sizeof(char));
      strcpy(code, S_look(variables, expression->u.variable->name));
      return code;
      break;
    case A_number_Exp:
      /* need to be work out, but 20 should do */
      temp = calloc(20, sizeof(char));
      sprintf(temp, "%d",expression->u.intt);
      code = calloc(strlen(temp) + 1, sizeof(char));
      strcpy(code, temp);
      free(temp);
      return code;
      break;

    case A_operator_Exp:
      temp = TRANS_expressioncode(expression->u.operator.left);
      temp2 = TRANS_expressioncode(expression->u.operator.right);
      temp3 = TRANS_operator(expression->u.operator.oper);

      code = concat("(", concat(temp, concat(" ", concat(temp3, concat(" ",
concat(temp2, ")"))))));

      free(temp);
      free(temp2);
      free(temp3);
      return code;
      break;

    case A_if_Exp:
      temp = TRANS_expressioncode(expression->u.iff.test);
      temp2 = TRANS_expressioncode(expression->u.iff.then);
      temp3 = TRANS_expressioncode(expression->u.iff.elsee);

      code = concat("(", concat(temp, concat(" ? ", concat(temp2, concat(" : ",
concat(temp3, ")"))))));

      free(temp);
      free(temp2);
      free(temp3);
      return code;
      break;

    case A_pair_Exp:
      temp = TRANS_expressioncode(expression->u.pair.left);
      temp2 = TRANS_expressioncode(expression->u.pair.right);
      temp3 = TRANS_typename(expression->type);

      code = concat(temp3, concat("_pair(", concat(temp, concat(", ",
concat(temp2, ")")))));

      free(temp);
      free(temp2);
      free(temp3);
      return code;
      break;

    case A_nil_Exp:
      temp = TRANS_typename(expression->type);

      code = concat(temp, "_nil()");

      free(temp);
      return code;
      break;

    case A_cons_Exp:
```

```
      temp = TRANS_expressioncode(expression->u.cons.a);
      temp2 = TRANS_expressioncode(expression->u.cons.b);
      temp3 = TRANS_expressioncode(expression->u.cons.c);
      temp4 = TRANS_typename(expression->type);

      code = concat(temp4, concat("_cons(", concat(temp, concat(", ",
concat(temp2, concat(", ", concat(temp3, ")")))))));

      free(temp);
      free(temp2);
      free(temp3);
      free(temp4);
      return code;
      break;

    case A_leaf_Exp:
      temp = TRANS_expressioncode(expression->u.leaf);
      temp2 = TRANS_typename(expression->type);

      code = concat(temp2, concat("_leaf(", concat(temp, ")")));

      free(temp);
      free(temp2);
      return code;
      break;

    case A_node_Exp:
      temp = TRANS_expressioncode(expression->u.node.a);
      temp2 = TRANS_expressioncode(expression->u.node.b);
      temp3 = TRANS_expressioncode(expression->u.node.c);
      temp4 = TRANS_expressioncode(expression->u.node.d);
      temp5 = TRANS_expressioncode(expression->u.node.e);
      temp6 = TRANS_typename(expression->type);

      code = concat(temp6, concat("_node(", concat(temp, concat(", ",
concat(temp2, concat(", ", concat(temp3, concat(", ", concat(temp4, concat(",
", concat(temp5, ")")))))))))));

      free(temp);
      free(temp2);
      free(temp3);
      free(temp4);
      free(temp5);
      free(temp6);
      return code;
      break;

    case A_function_call_Exp:
      temp = calloc(strlen(S_name(expression->u.function_call.name)) + 1,
sizeof(char));
      strcpy(temp, S_name(expression->u.function_call.name));
      temp2 = TRANS_expression_listcode(expression->u.function_call.arguments);

      code = concat(temp, concat("(", concat(temp2, ")")));

      free(temp2);
      free(temp);
      return code;
      break;

    case A_match_pair_Exp:
      /* create variables */
      temp = VAR_fresh();
      temp4 = concat(temp, ".fst");
      temp5 = concat(temp, ".snd");

      /* put fresh variable in declarations */
      temp6 = concat(declarations,
concat(TRANS_typename(expression->u.match_pair.a->type), concat(" ",
concat(temp, ";\n"))));
      free(declarations);
      declarations = temp6;

      /* nested variable scope */
      temp2 = TRANS_expressioncode(expression->u.match_pair.a);
      S_beginScope(variables);
      S_enter(variables, expression->u.match_pair.b->name, temp4);
      S_enter(variables, expression->u.match_pair.c->name, temp5);
      temp3 = TRANS_expressioncode(expression->u.match_pair.d);
      S_endScope(variables);
      free(temp4);
      free(temp5);

      /* put it all together */
      code = concat("(", concat(temp, concat("=", concat(temp2, concat(", ",
concat(temp3, ")"))))));
      free(temp);
      free(temp2);
```

```
      free(temp3);
      return code;
      break;

    case A_match_list_Exp:
      temp = VAR_fresh();

      temp6 = concat(temp, ".d");
      temp7 = concat(temp, ".hd");
      temp8 = concat(temp, ".tl");

      temp9 = concat(declarations,
concat(TRANS_typename(expression->u.match_list.a->type), concat("_destr_t ",
concat(temp, ";\n"))));

      free(declarations);
      declarations = temp9;


      temp2 = TRANS_typename(expression->u.match_list.a->type);
      temp3 = TRANS_expressioncode(expression->u.match_list.a);
      temp4 = TRANS_expressioncode(expression->u.match_list.b);
      S_beginScope(variables);
      S_enter(variables, expression->u.match_list.c->name, temp6);
      S_enter(variables, expression->u.match_list.d->name, temp7);
      S_enter(variables, expression->u.match_list.e->name, temp8);
      temp5 = TRANS_expressioncode(expression->u.match_list.f);
      S_endScope(variables);
      free(temp6);
      free(temp7);
      free(temp8);

      code = concat("(", concat(temp, concat("=", concat(temp2,
concat("_destr(", concat(temp3, concat("), ", concat(temp, concat(".kind == NIL
? ", concat(temp4, concat(" : ", concat(temp5, ")"))))))))))));

      free(temp);
      free(temp2);
      free(temp3);
      free(temp4);
      free(temp5);
      return code;
      break;

    case A_match_tree_Exp:
      temp = VAR_fresh();

      temp6 = concat(temp, ".left");
      temp7 = concat(temp, ".right");
      temp8 = concat(temp, ".label");
      temp9 = concat(temp, ".d1");
      temp10 = concat(temp, ".d2");

      temp11 = concat(declarations,
concat(TRANS_typename(expression->u.match_tree.a->type), concat("_destr_t ",
concat(temp, ";\n"))));

      free(declarations);
      declarations = temp11;

      temp2 = TRANS_typename(expression->u.match_tree.a->type);
      temp3 = TRANS_expressioncode(expression->u.match_tree.a);
      S_beginScope(variables);
      S_enter(variables, expression->u.match_tree.b->name, temp8);
      temp4 = TRANS_expressioncode(expression->u.match_tree.c);
      S_endScope(variables);
      S_beginScope(variables);
      S_enter(variables, expression->u.match_tree.d->name, temp9);
      S_enter(variables, expression->u.match_tree.e->name, temp10);
      S_enter(variables, expression->u.match_tree.f->name, temp8);
      S_enter(variables, expression->u.match_tree.g->name, temp6);
      S_enter(variables, expression->u.match_tree.h->name, temp7);
      temp5 = TRANS_expressioncode(expression->u.match_tree.i);
      S_endScope(variables);
      free(temp6);
      free(temp7);
      free(temp8);
      free(temp9);
      free(temp10);

      code = concat("(", concat(temp, concat("=", concat(temp2,
concat("_destr(", concat(temp3, concat("), ", concat(temp, concat(".kind ==
LEAF ? ", concat(temp4, concat(" : ", concat(temp5, ")"))))))))))));

      free(temp);
      free(temp2);
      free(temp3);
      free(temp4);
```

```
      free(temp5);
      return code;
      break;
    default:
      assert(0);
    }
}

char *TRANS_variablecode(A_variable variable)
{
  char *code;
  char *temp;

  S_enter(variables, variable->name, S_name(variable->name));
  temp = TRANS_typename(variable->type);

  code = concat(temp, concat(" ", S_name(variable->name)));

  free(temp);
  return code;
}

char *TRANS_variable_listcode(A_variable_list variable_list)
{
  char *temp;
  char *temp2;
  char *code;

  if (variable_list != NULL)
    {
      temp = TRANS_variablecode(variable_list->head);
      temp2 = TRANS_variable_listcode(variable_list->tail);
      code = calloc(strlen(temp) + strlen(temp2) + 3, sizeof(char));
      strcpy(code, temp);
      if (strlen(temp2) != 0)
{
  strcat(code, ", ");
  strcat(code, temp2);
}
      free(temp);
      free(temp2);
      return code;
    }
  else
    {
      code = calloc(1, sizeof(char));
      strcpy(code, "");
      return code;
    }
}

void TRANS_definitioncode(A_definition definition)
{
  char *temp;
  char *temp2;
  char *temp3;
  char *temp5;
  char *code;

  VAR_reset();
  S_beginScope(variables);
  temp = calloc(strlen(S_name(definition->name)) + 1, sizeof(char));
  strcpy(temp, S_name(definition->name));
  temp2 = TRANS_variable_listcode(definition->parameter_list);
  temp3 = TRANS_expressioncode(definition->body);
  temp5 = TRANS_typename(definition->body->type);
  S_endScope(variables);

  /* if empty list place in void */
  if (strlen(temp2) == 0)
    {
      free(temp2);
      temp2 = calloc(5, sizeof(char));
      strcpy(temp2, "void");
    }

  code = concat(functions, concat(temp5, concat(" ", concat(temp, concat("(",
concat(temp2, concat(")\n{\n", concat(declarations, concat("\nreturn ",
concat(temp3, ";\n}\n\n"))))))))));

  free(declarations);
  declarations = calloc(1, sizeof(char));
  strcpy(declarations, "");
  free(temp);
  free(temp2);
  free(temp3);
  free(functions);
  functions = code;
```

```
}

char *TRANS_type_listcode(A_type_list type_list)
{
  char *temp;
  char *temp2;
  char *temp3;
  char *temp4;
  char *code;

  if (type_list != NULL)
    {
      /* add parameter types to global type list */
      temp3 = TRANS_recursive_typecode(type_list->head);
      temp4 = calloc(strlen(types) + strlen(temp3) + 1, sizeof(char));
      strcpy(temp4, types);
      strcat(temp4, temp3);
      free(types);
      free(temp3);
      types = temp4;

      /* create parameter list */
      temp = TRANS_typename(type_list->head);
      temp2 = TRANS_type_listcode(type_list->tail);
      code = calloc(strlen(temp) + strlen(temp2) + 3, sizeof(char));
      strcpy(code, temp);
      if (strlen(temp2))
{
  strcat(code, ", ");
  strcat(code, temp2);
}
      free(temp);
      free(temp2);
      return code;
    }
  else
    {
      code = calloc(1, sizeof(char));
      strcpy(code, "");
      return code;
    }
}

void TRANS_signaturecode(A_signature signature)
{

  char *code;
  char *parameters;
  char *result;
  char *name;
  char *temp;
  char *temp2;

  result = TRANS_typename(signature->result);
  name = calloc(strlen(S_name(signature->name)) + 1, sizeof(char));
  strcpy(name, S_name(signature->name));

  temp = TRANS_recursive_typecode(signature->result);
  temp2 = concat(types, temp);

  free(types);
  free(temp);
  types = temp2;

  parameters = TRANS_type_listcode(signature->parameter_type_list);

  if (strlen(parameters) == 0)
    {
      free(parameters);
      parameters = calloc(5, sizeof(char));
      strcpy(parameters, "void");
    }

  code = concat(prototypes, concat(result, concat(" ", concat(name, concat("(",
concat(parameters, ");\n\n"))))));

  free(name);
  free(prototypes);
  free(parameters);
  free(result);
  prototypes = code;
}

void TRANS_signature_definitioncode(A_signature_definition signature_definition)
{
  switch (signature_definition->kind)
    {
```

```
    case A_sig:
      TRANS_signaturecode(signature_definition->u.sig);
      break;
    case A_def:
      TRANS_definitioncode(signature_definition->u.def);
      break;
    default:
      assert(0);
    }
}

void TRANS_sig_def_listcode(A_sig_def_list sig_def_list)
{
  if (sig_def_list != NULL)
    {
      TRANS_signature_definitioncode(sig_def_list->head);
      TRANS_sig_def_listcode(sig_def_list->tail);
    }
  else
    {
      /* do nothing */
    }
}

char *TRANS_programcode(A_program program)
{
  char *code;
  if (program != NULL)
    {
      variables = S_empty();
      types = calloc(87, sizeof(char));
      prototypes = calloc(30 , sizeof(char));
      functions = calloc(30 , sizeof(char));
      declarations = calloc(1, sizeof(char));
      strcpy(types,  "/* TYPE DEFINITIONS */\ntypedef enum {INL, INR, NIL,
CONS, LEAF, NODE} kind_t;\n\n");
      strcpy(prototypes, "/* FUNCTION PROTOTYPES */\n");
      strcpy(functions, "/* FUNCTION DEFINITIONS */\n");
      strcpy(declarations, "");

      TRANS_sig_def_listcode(program->sig_def_list);

      code = concat(types, concat("\n", concat(prototypes, concat("\n",
functions))));

      fprintf(stderr, "Code Generation successful!\n");
      return code;
    }
  else
    {
      code = "";
      fprintf(stderr, "Code Generation successful!\n");
      return code;
    }
}
```

A.21 types.h

```
/* type.h */

typedef struct Ty_function_ *Ty_function;

A_type Ty_Int(void);
A_type Ty_Diamond(void);
A_type Ty_List(A_type list_type);
A_type Ty_Tree(A_type tree_type);
A_type Ty_Pair(A_type left, A_type right);

struct Ty_function_
{
  A_type_list formals;
  A_type result;
};

Ty_function Ty_Function(A_type_list formals, A_type result);


int Ty_are_equal(A_type a, A_type b);

int Ty_is_heap(A_type a);

void Ty_type_print(A_type t);
```

```
void Ty_type_list_print(A_type_list list);
```

A.22 types.c

```
/* type.c */

#include <stdio.h>
#include "util.h"
#include "symbol.h"
#include "abstractsyntax.h"
#include "types.h"

static struct A_type_ tyint = {A_int_Type};

A_type Ty_Int(void)
{
return &tyint;
}

static struct A_type_ tydiamond = {A_diamond_Type};

A_type Ty_Diamond(void)
{
return &tydiamond;
}

A_type Ty_List(A_type list_type)
{
  A_type p = (A_type)checked_malloc(sizeof(*p));
  p->kind = A_list_Type;
  p->u.list_type = list_type;
  return p;
}

A_type Ty_Tree(A_type tree_type)
{
  A_type p = (A_type)checked_malloc(sizeof(*p));
  p->kind = A_tree_Type;
  p->u.tree_type = tree_type;
  return p;
}

A_type Ty_Pair(A_type left, A_type right)
{
  A_type p = (A_type)checked_malloc(sizeof(*p));
  p->kind = A_pair_Type;
  p->u.pair.left = left;
  p->u.pair.right = right;
  return p;
}


Ty_function Ty_Function(A_type_list formals, A_type result)
{
  Ty_function p = (Ty_function)checked_malloc(sizeof(*p));
  p->formals = formals;
  p->result = result;
  return p;
}

int Ty_is_heap(A_type a)
{
  switch (a->kind)
    {
    case A_int_Type:
      return FALSE;
    case A_diamond_Type:
      return TRUE;
    case A_list_Type:
      return TRUE;
    case A_tree_Type:
      return TRUE;
    case A_pair_Type:
      return (Ty_is_heap(a->u.pair.left) || Ty_is_heap(a->u.pair.right));
    default:
      assert(0);
    }
}

int Ty_are_equal(A_type a, A_type b)
{
  switch (a->kind)
    {
```

```
    case A_int_Type:
      if (b->kind == A_int_Type)
{
  return TRUE;
}
      else
{
  return FALSE;
}
    case A_diamond_Type:
      if (b->kind == A_diamond_Type)
{
  return TRUE;
}
      else
{
  return FALSE;
}
    case A_list_Type:
      if (b->kind == A_list_Type)
{
  return (Ty_are_equal(a->u.list_type, b->u.list_type));
}
      else
{
  return FALSE;
}
    case A_tree_Type:
      if (b->kind == A_tree_Type)
{
  return (Ty_are_equal(a->u.tree_type, b->u.tree_type));
}
      else
{
  return FALSE;
}
    case A_pair_Type:
      if (b->kind == A_pair_Type)
{
  return (Ty_are_equal(a->u.pair.left, b->u.pair.left) &&
Ty_are_equal(a->u.pair.right, b->u.pair.right));
}
      else
{
  return FALSE;
}
    default:
      assert(0);
    }
}

static char string_types[][15] = {"int", "diamond", "list", "tree", "pair"};

void Ty_type_print(A_type t)
{
  if (t == NULL)
    {
      fprintf(stderr, "null");
    }
  else
    {
      switch (t->kind)
        {
        case A_int_Type:
          fprintf(stderr, "%s", string_types[t->kind]);
          break;
        case A_diamond_Type:
          fprintf(stderr, "%s", string_types[t->kind]);
          break;
        case A_list_Type:
          fprintf(stderr, "%s(", string_types[t->kind]);
          Ty_type_print(t->u.list_type);
          fprintf(stderr, ")");
          break;
        case A_tree_Type:
          fprintf(stderr, "%s(", string_types[t->kind]);
          Ty_type_print(t->u.tree_type);
          fprintf(stderr, ")");
          break;
        case A_pair_Type:
          fprintf(stderr, "%s(", string_types[t->kind]);
          Ty_type_print(t->u.pair.left);
          fprintf(stderr, ",");
          Ty_type_print(t->u.pair.right);
          fprintf(stderr, ")");
          break;
        default:
          assert(0);
```

```
        }
    }
}

void Ty_type_list_print(A_type_list list)
{
  if (list == NULL)
    {
      fprintf(stderr, "null");
    }
  else
    {
      fprintf(stderr, "Ty_List(");
      Ty_type_print(list->head);
      fprintf(stderr, ", ");
      Ty_type_list_print(list->tail);
      fprintf(stderr, ")");
    }
}
```

A.23 util.h

```
#include <assert.h>

typedef char *string;
typedef char bool;

#define TRUE 1
#define FALSE 0

void *checked_malloc(int);
string String(char *);

typedef struct U_boolList_ *U_boolList;
struct U_boolList_ {bool head; U_boolList tail;};
U_boolList U_BoolList(bool head, U_boolList tail);
```

A.24 util.c

```
/*
 * util.c - commonly used utility functions.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "util.h"
void *checked_malloc(int len)
{void *p = malloc(len);
 if (!p) {
    fprintf(stderr,"\nRan out of memory!\n");
    exit(1);
 }
 return p;
}

string String(char *s)
{string p = checked_malloc(strlen(s)+1);
 strcpy(p,s);
 return p;
}

U_boolList U_BoolList(bool head, U_boolList tail)
{ U_boolList list = checked_malloc(sizeof(*list));
  list->head = head;
  list->tail = tail;
  return list;
}
```

A.25 variables.h

```
char *VAR_fresh(void);

void VAR_reset(void);
```

A.26 variables.c

```
#include <stdlib.h>
#include <stdio.h>

static int VAR_counter = 0;

char *VAR_fresh(void)
{
  char *temp;
  temp = calloc(20, sizeof(char));
  sprintf(temp, "x%d",VAR_counter);
  VAR_counter++;
  return temp;
}

void VAR_reset(void)
{
  VAR_counter = 0;
}
```

A.27 Makefile

```
#
# makefile for functional compiler
#

FLAGS = -g -O -Wall

compiler: main.o fun.grm.tab.o lex.yy.o errormsg.o util.o abstractsyntax.o symbol.o table.o prettyprint.o types.o semantic.o sets.o translation.o variables.o
gcc $(FLAGS) -o compiler main.o fun.grm.tab.o lex.yy.o errormsg.o util.o abstractsyntax.o symbol.o table.o prettyprint.o types.o semantic.o sets.o translation.o

main.o: main.c parse.h abstractsyntax.h symbol.h prettyprint.h errormsg.h util.h semantic.h translation.h
gcc $(FLAGS) -c main.c

fun.grm.tab.o: fun.grm.tab.c
gcc $(FLAGS) -c fun.grm.tab.c

fun.grm.tab.c: fun.grm
bison -dv fun.grm

fun.grm.tab.h: fun.grm.tab.c
echo "fun.grm.tab.h created when fun.grm.tab.c was created"

errormsg.o: errormsg.c errormsg.h util.h
gcc $(FLAGS) -c errormsg.c

lex.yy.o: lex.yy.c fun.grm.tab.h errormsg.h util.h
gcc $(FLAGS) -c lex.yy.c

lex.yy.c: fun.lex
flex fun.lex

abstractsyntax.o: abstractsyntax.c util.h symbol.h abstractsyntax.h
gcc $(FLAGS) -c abstractsyntax.c

symbol.o: symbol.c symbol.h util.h table.h
gcc $(FLAGS) -c symbol.c

table.o: table.c table.h util.h
gcc $(FLAGS) -c table.c

util.o: util.c util.h
gcc $(FLAGS) -c util.c

prettyprint.o: prettyprint.c prettyprint.h util.h symbol.h abstractsyntax.h
gcc $(FLAGS) -c prettyprint.c

types.o: types.c types.h util.h abstractsyntax.h symbol.h
gcc $(FLAGS) -c types.c

semantic.o: semantic.c semantic.h util.h errormsg.h symbol.h abstractsyntax.h types.h sets.h
gcc $(FLAGS) -c semantic.c

sets.o: sets.c util.h errormsg.h symbol.h
gcc $(FLAGS) -c sets.c

translation.o: translation.c util.h errormsg.h types.h symbol.h sets.h abstractsyntax.h variables.h
gcc $(FLAGS) -c translation.c

variables.o: variables.c
gcc $(FLAGS) -c variables.c
```

```
clean:
rm -f *.o fun.grm.tab.c fun.grm.tab.h lex.yy.c fun.grm.output *~


abstractsyntax.h: symbol.h
```