Custom EBook 4

Written by Jainish Parmar

Table of Contents

5.6 Further Reading

As its name suggests, YACC was not the first compiler construction tool, but it remains widely used today and has led to a proliferation of similar tools written in various languages and addressing different classes of grammars. Here is just a small selection:

1. S. C. Johnson, "YACC: Yet Another Compiler-Compiler", Bell Laboratories Technical Journal, 1975.

2. D. Grune and C.J.H Jacobs, "A programmer-friendly LL(1) parser generator", Software: Practice and Experience, volume 18, number 1.

3. T.J. Parr and R.W. Quong, "ANTLR: A predicated LL(k) Parser Generator", Software: Practice and Experience, 1995.

4. S. McPeak, G.C. Necula, "Elkhound: A Fast, Practical GLR Parser Generator", International Conference on Compiler Construction, 2004.

Chapter 6 – The Abstract Syntax Tree

6.1 Overview

The **abstract syntax tree (AST)** is an important internal data structure that represents the primary structure of a program. The AST is the starting point for semantic analysis of a program. It is "abstract" in the sense that the structure leaves out the particular details of parsing: the AST does not care whether a language has prefix, postfix, or infix expressions. (In fact, the AST we describe here can be used to represent most procedural languages.)

For our project compiler, we will define an AST in terms of five C structures representing declarations, statements, expressions, types, and parameters. While you have certainly encountered each of these terms while learning programming, they are not always used precisely in practice. This chapter will help you to sort those items out very clearly:

- A **declaration** states the name, type, and value of a symbol so that it can be used in the program. Symbols include items such as constants, variables, and functions.
- A **statement** indicates an action to be carried out that changes the state of the program. Examples include loops, conditionals, and function returns.
- An **expression** is a combination of values and operations that is **evaluated** according to specific rules and yields a **value** such as an integer, floating point, or string. In some programming languages, an expression may also have a **side effect** that changes the state of the program.

For each kind of element in the AST, we will give an example of the code and how it is constructed. Because each of these structures potentially has pointers to each of the other types, it is necessary to preview all of them before seeing how they work together.

Once you understand all of the elements of the AST, we finish the chapter by demonstrating how the entire structure can be created automatically through the use of the Bison parser generator.

6.2 Declarations

A complete B-Minor program is a sequence of declarations. Each declaration states the existence of a variable or a function. A variable declaration may optionally give an initializing value. If none is given, it is given a default value of zero. A function declaration may optionally give the body of the function in code; if no body is given, then the declaration serves as a prototype for a function declared elsewhere.

For example, the following are all valid declarations:

```
b: boolean;
s: string = "hello";
f: function integer ( x: integer ) = { return x*x; }
```

A declaration is represented by a `decl` structure that gives the name, type, value (if an expression), code (if a function), and a pointer to the next declaration in the program:

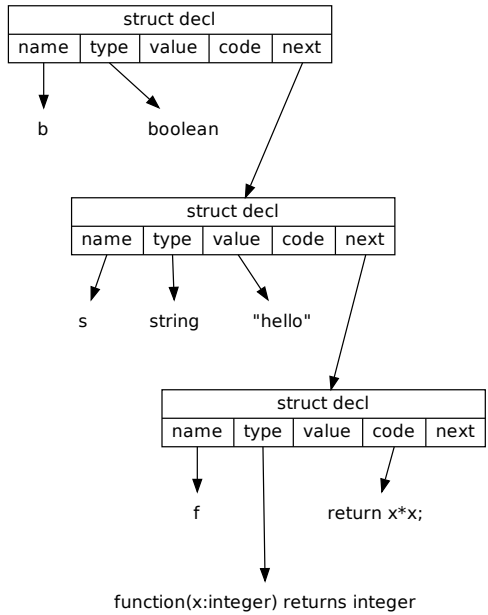
```
struct decl {
    char *name;
    struct type *type;
    struct expr *value;
    struct stmt *code;
    struct decl *next;
};
```

Because we will be creating a lot of these structures, you will need a factory function that allocates a structure and initializes its fields, like this:

```
struct decl * decl_create( char *name,
                           struct type *type,
                           struct expr *value,
                           struct stmt *code,
                           struct decl *next )
{
    struct decl *d = malloc(sizeof(*d));
    d->name = name;
    d->type = type;
    d->value = value;
    d->code = code;
    d->next = next;
    return d;
}
```

(You will need to write similar code for statements, expressions, etc, but we won't keep repeating it here.)

The three declarations on the preceding page can be represented graphically as a linked list, like this:



Note that some of the fields point to nothing: these would be represented by a null pointer, which we omit for clarity. Also, our picture is incomplete and must be expanded: the items representing types, expressions, and statements are all complex structures themselves that we must describe.

6.3 Statements

The body of a function consists of a sequence of statements. A statement indicates that the program is to take a particular action in the order specified, such as computing a value, performing a loop, or choosing between branches of an alternative. A statement can also be a declaration of a local variable. Here is the `stmt` structure:

```
struct stmt {
    stmt_t kind;
    struct decl *decl;
    struct expr *init_expr;
    struct expr *expr;
    struct expr *next_expr;
    struct stmt *body;
    struct stmt *else_body;
    struct stmt *next;
};
```

```
typedef enum {
    STMT_DECL,
    STMT_EXPR,
    STMT_IF_ELSE,
    STMT_FOR,
    STMT_PRINT,
    STMT_RETURN,
    STMT_BLOCK
} stmt_t;
```

The `kind` field indicates what kind of statement it is:

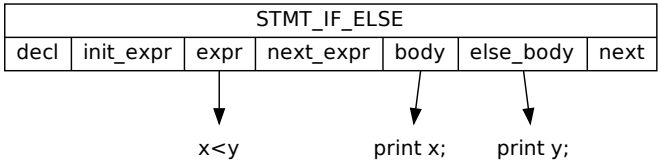
- `STMT_DECL` indicates a (local) declaration, and the `decl` field will point to it.
- `STMT_EXPR` indicates an expression statement and the `expr` field will point to it.
- `STMT_IF_ELSE` indicates an if-else expression such that the `expr` field will point to the control expression, the `body` field to the statements executed if it is true, and the `else_body` field to the statements executed if it is false.
- `STMT_FOR` indicates a for-loop, such that `init_expr`, `expr`, and `next_expr` are the three expressions in the loop header, and `body` points to the statements in the loop.
- `STMT_PRINT` indicates a `print` statement, and `expr` points to the expressions to print.
- `STMT_RETURN` indicates a `return` statement, and `expr` points to the expression to return.
- `STMT_BLOCK` indicates a block of statements inside curly braces, and `body` points to the contained statements.

And, as we did with declarations, we require a function `stmt_create` to create and return a statement structure:

```
struct stmt * stmt_create( stmt_t kind,
    struct decl *decl, struct expr *init_expr,
    struct expr *expr, struct expr *next_expr,
    struct stmt *body, struct stmt *else_body,
    struct stmt *next );
```

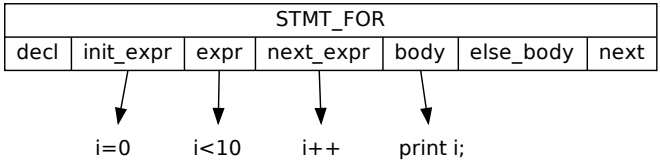
This structure has a lot of fields, but each one serves a purpose and is used when necessary for a particular kind of statement. For example, an if-else statement only uses the `expr`, `body`, and `else_body` fields, leaving the rest null:

```
if( x<y ) print x; else print y;
```



A for-loop uses the three `expr` fields to represent the three parts of the loop control, and the `body` field to represent the code being executed:

```
for(i=0;i<10;i++) print i;
```



6.4 Expressions

Expressions are implemented much like the simple expression AST shown in Chapter 5. The difference is that we need many more binary types: one for every operator in the language, including arithmetic, logical, comparison, assignment, and so forth. We also need one for every type of leaf value, including variable names, constant values, and so forth. The `name` field will be set for `EXPR_NAME`, the `integer_value` field for `EXPR_INTEGER_LITERAL`, and so on. You may need to add values and types to this structure as you expand your compiler.

```
struct expr {
    expr_t kind;
    struct expr *left;
    struct expr *right;

    const char *name;
    int integer_value;
    const char *
       string_literal;
};
```

```
typedef enum {
    EXPR_ADD,
    EXPR_SUB,
    EXPR_MUL,
    EXPR_DIV,
    ...
    EXPR_NAME,
    EXPR_INTEGER_LITERAL,
    EXPR_STRING_LITERAL
} expr_t;
```

As before, you should create a factory for a binary operator:

```
struct expr * expr_create( expr_t kind,
                           struct expr *L,
                           struct expr *R );
```

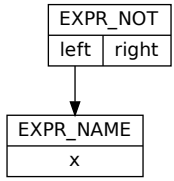
And then a factory for each of the leaf types:

```
struct expr * expr_create_name( const char *name );
struct expr * expr_create_integer_literal( int i );
struct expr * expr_create_boolean_literal( int b );
struct expr * expr_create_char_literal( char c );
struct expr * expr_create_string_literal
                               ( const char *str );
```

Note that you can store the integer, boolean, and character literal values all in the `integer_value` field.

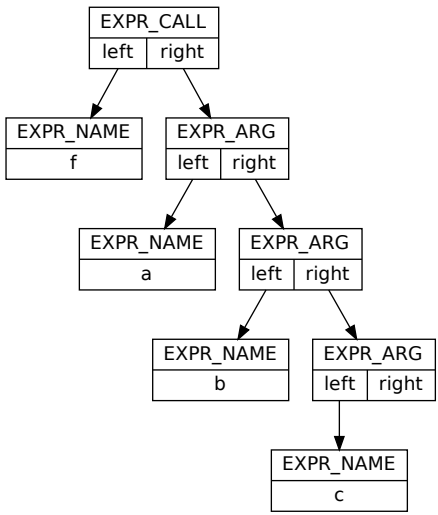
A few cases deserve special mention. Unary operators like logical-not typically have their sole argument in the `left` pointer:

```
!x
```



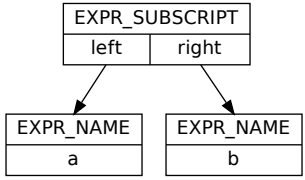
A function call is constructed by creating an `EXPR_CALL` node, such that the left-hand side is the function name, and the right hand side is an unbalanced tree of `EXPR_ARG` nodes. While this looks a bit awkward, it allows us to express a linked list using a tree, and will simplify the handling of function call arguments on the stack during code generation.

```
f(a,b,c)
```



Array subscripting is treated like a binary operator, such that the name of the array is on the left side of the `EXPR_SUBSCRIPT` operator, and an integer expression on the right:

```
a[b]
```



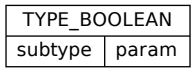
6.5 Types

A `type` structure encodes the type of every variable and function mentioned in a declaration. Primitive types like `integer` and `boolean` are expressed by simply setting the `kind` field appropriately, and leaving the other fields null. Compound types like `array` and `function` are built by connecting multiple `type` structures together.

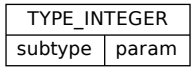
```
typedef enum {
    TYPE_VOID,
    TYPE_BOOLEAN,
    TYPE_CHARACTER,
    TYPE_INTEGER,
    TYPE_STRING,
    TYPE_ARRAY,
    TYPE_FUNCTION
} type_t;

struct type {
    type_t kind;
    struct type *subtype;
    struct param_list *params;
};

struct param_list {
    char *name;
    struct type *type;
    struct param_list *next;
};
```

For example, to express a basic type like a boolean or an integer, we simply create a standalone `type` structure, with `kind` set appropriately, and the other fields null:

`boolean`

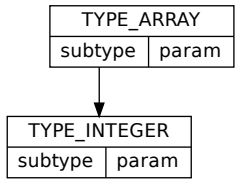


`integer`



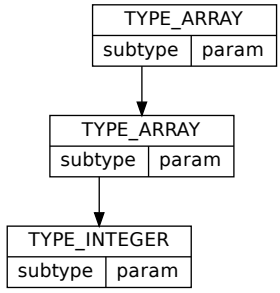
To express a compound type like an array of integers, we set `kind` to `TYPE_ARRAY` and set `subtype` to point to a `TYPE_INTEGER`:

`array [] integer`



These can be linked to arbitrary depth, so to express an array of array of integers:

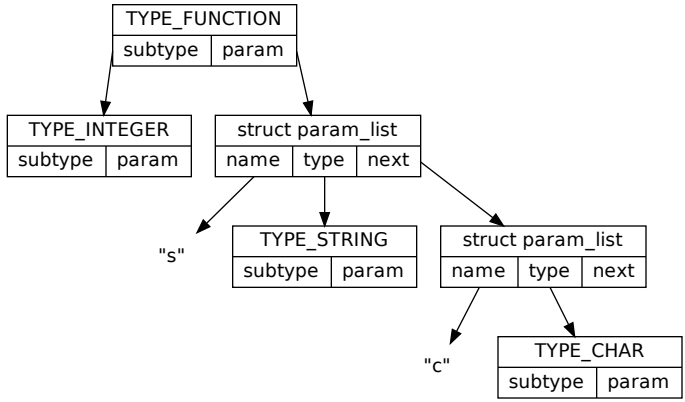
`array [] array [] integer`



To express the type of a function, we use `subtype` to express the return type of the function, and then connect a linked list of `param_list` nodes to describe the name and type of each parameter to the function.

For example, here is the type of a function which takes two arguments and returns an integer:

```
function integer (s:string, c:char)
```



Note that the type structures here let us express some complicated and powerful higher order concepts of programming. By simply swapping in complex types, you can describe an array of ten functions, each returning an integer:

```
a: array [10] function integer ( x: integer );
```

Or how about a function that returns a function?

```
f: function function integer (x:integer) (y:integer);
```

Or even a function that returns an array of functions!

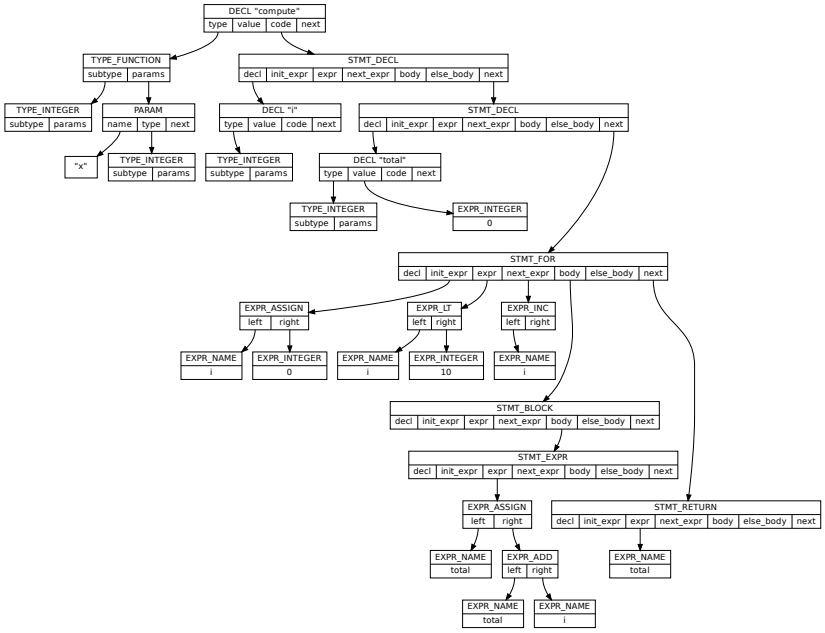
```
g: function array [10]
       function integer (x:integer) (y:integer);
```

While the B-Minor type system is capable of *expressing* these ideas, these combinations will be rejected later in typechecking, because they require a more dynamic implementation than we are prepared to create. If you find these ideas interesting, then you should read up on functional languages such as Scheme and Haskell.

6.6 Putting it All Together

Now that you have seen each individual component, let's see how a complete B-Minor function would be expressed as an AST:

```
compute: function integer ( x:integer ) = {
        i: integer;
        total: integer = 0;
        for(i=0;i<10;i++) {
                total = total + i;
        }
        return total;
}
```



6.7 Building the AST

With the functions created so far in this chapter, we could, in principle, construct the AST manually in a sort of nested style. For example, the following code represents a function called `square` which accepts an integer `x` as a parameter, and returns the value `x*x`:

```
d = decl_create(
    "square",
    type_create(TYPE_FUNCTION,
        type_create(TYPE_INTEGER,0,0),
        param_list_create(
            "x",
            type_create(TYPE_INTEGER,0,0),
            0)),
    0,
    stmt_create(STMT_RETURN,0,0,
        expr_create(EXPR_MUL,
            expr_create_name("x"),
            expr_create_name("x")),
        0,0,0,0),
    0);
```

Obviously, this is no way to write code! Instead, we want our parser to invoke the various creation functions whenever they are reduced, and then hook them up into a complete tree. Using an LR parser generator like Bison, this process is straightforward. (Here I will give you the idea of how to proceed, but you will need to figure out many of the details in order to complete the parser.)

At the top level, a B-Minor program is a sequence of declarations:

```
program : decl_list
          { parser_result = $1; }
        ;
```

Then, we write rules for each of the various kinds of declarations in a B-Minor program:

```
decl : name TOKEN_COLON type TOKEN_SEMI
     { $$ = decl_create($1,$3,0,0,0); }
   | name TOKEN_COLON type TOKEN_ASSIGN expr TOKEN_SEMI
     { $$ = decl_create($1,$3,$5,0,0); }
   | /* and more cases here */
       . . .
   ;
```

Since each `decl` structure is created separately, we must connect them together in a linked list formed by a `decl_list`. This is most easily done by making the rule right-recursive, so that `decl` on the left represents one declaration, and `decl_list` on the right represents the remainder of the linked list. The end of the list is a null value when `decl_list` produces ϵ.

```
decl_list : decl decl_list
            { $$ = $1; $1->next = $2; }
          | /* epsilon */
            { $$ = 0; }
          ;
```

For each kind of statement, we create a `stmt` structure that pulls out the necessary elements from the grammar.

```
stmt : TOKEN_IF TOKEN_LPAREN expr TOKEN_RPAREN stmt
       { $$ = stmt_create(STMT_IF_ELSE,0,0,$3,0,$5,0,0); }
     | TOKEN_LBRACE stmt_list TOKEN_RBRACE
       { $$ = stmt_create(STMT_BLOCK,0,0,0,0,$2,0,0); }
     | /* and more cases here */
       . . .
     ;
```

Proceed in this way down through each of the grammar elements of a B-Minor program: declarations, statements, expressions, types, parameters, until you reach the leaf elements of literal values and symbols, which are handled in the same way as in Chapter 5.

There is one last complication: What, exactly is the semantic type of the values returned as each rule is reduced? It isn't a single type, because each kind of rule returns a different data structure: a declaration rule returns a `struct decl *`, while an identifier rule returns a `char *`. To make this work, we inform Bison that the semantic value is the union of all of the types in the AST:

```
%union {
        struct decl *decl;
        struct stmt *stmt;
        . . .
        char *name;
};
```

And then indicate the specific subfield of the union used by each rule:

```
%type <decl> program decl_list decl . . .
%type <stmt> stmt_list stmt . . .
. . .
%type <name> name
```

6.8 Exercises

1. Write a complete LR grammar for B-Minor and test it using Bison. Your first attempt will certainly have many shift-reduce and reduce-reduce conflicts, so use your knowledge of grammars from Chapter 4 to rewrite the grammar and eliminate the conflicts.

2. Write the AST structures and generating functions as outlined in this chapter, and manually construct some simple ASTs using nested function calls as shown above.

3. Add new functions `decl_print()`, `stmt_print()`, etc. that print the AST back out so you can verify that the program was generated correctly. Make your output nicely formatted using indentation and consistent spacing, so that the code is easily readable.

4. Add the AST generator functions as action rules to your Bison grammar, so that you can parse complete programs, and print them back out again.

5. Add new functions `decl_translate()`, `stmt_translate()`, etc. that output the B-Minor AST in a different language of your own choosing, such as Python or Java or Rust.

6. Add new functions that emit the AST in a graphical form so you can "see" the structure of a program. One approach would be to use the Graphviz DOT format: let each declaration, statement, etc be a node in a graph, and then let each pointer between structures be an edge in the graph.

Chapter 7 – Semantic Analysis

Now that we have completed construction of the AST, we are ready to begin analyzing the **semantics**, or the actual meaning of a program, and not simply its structure.

Type checking is a major component of semantic analysis. Broadly speaking, the type system of a programming language gives the programmer a way to make verifiable assertions that the compiler can check automatically. This allows for the detection of errors at compile-time, instead of at runtime.

Different programming languages have different approaches to type checking. Some languages (like C) have a rather weak type system, so it is possible to make serious errors if you are not careful. Other languages (like Ada) have very strong type systems, but this makes it more difficult to write a program that will compile at all!

Before we can perform type checking, we must determine the type of each identifier used in an expression. However, the mapping between variable names and their actual storage locations is not immediately obvious. A variable `x` in an expression could refer to a local variable, a function parameter, a global variable, or something else entirely. We solve this problem by performing **name resolution**, in which each definition of a variable is entered into a **symbol table**. This table is referenced throughout the semantic analysis stage whenever we need to evaluate the correctness of some code.

Once name resolution is completed, we have all the information necessary to check types. In this stage, we compute the type of complex expressions by combining the basic types of each value according to standard conversion rules. If a type is used in a way that is not permitted, the compiler will output an (ideally helpful) error message that will assist the programmer in resolving the problem.

Semantic analysis also includes other forms of checking the correctness of a program, such as examining the limits of arrays, avoiding bad pointer traversals, and examining control flow. Depending on the design of the language, some of these problems can be detected at compile time, while others may need to wait until runtime.

7.1 Overview of Type Systems

Most programming languages assign to every value (whether a literal, constant, or variable) a **type**, which describes the interpretation of the data in that variable. The type indicates whether the value is an integer, a floating point number, a boolean, a string, a pointer, or something else. In most languages, these atomic types can be combined into higher-order types such as enumerations, structures, and variant types to express complex constraints.

The type system of a language serves several purposes:

- **Correctness.** A compiler uses type information provided by the programmer to raise warnings or errors if a program attempts to do something improper. For example, it is almost certainly an error to assign an integer value to a pointer variable, even though both might be implemented as a single word in memory. A good type system can help to eliminate runtime errors by flagging them at compile time instead.

- **Performance.** A compiler can use type information to find the most efficient implementation of a piece of code. For example, if the programmer tells the compiler that a given variable is a constant, then the same value can be loaded into a register and used many times, rather than constantly loading it from memory.

- **Expressiveness.** A program can be made more compact and expressive if the language allows the programmer to leave out facts that can be inferred from the type system. For example, in B-Minor the `print` statement does not need to be told whether it is printing an integer, a string, or a boolean: the type is inferred from the expression and the value is automatically displayed in the proper way.

A programming language (and its type system) are commonly classified on the following axes:

- safe or unsafe
- static or dynamic
- explicit or implicit

In an **unsafe programming language**, it is possible to write valid programs that have wildly undefined behavior that violates the basic structure of the program. For example, in the C programming language, a program can construct an arbitrary pointer to modify any word in memory, and thereby change the data and code of the compiled program. Such power is probably necessary to implement low-level code like an operating system or a driver, but is problematic in general application code.

For example, the following code in C is syntactically legal and will compile, but is unsafe because it writes data outside the bounds of the array `a[]`. As a result, the program could have almost any outcome, including incorrect output, silent data corruption, or an infinite loop.

```
/* This is C code */
int i;
int a[10];
for(i=0;i<100;i++) a[i] = i;
```

In a **safe programming language**, it is not possible to write a program that violates the basic structures of the language. That is, no matter what input is given to a program written in a safe language, it will always execute in a well-defined way that preserves the abstractions of the language. A safe programming language enforces the boundaries of arrays, the use of pointers, and the assignment of types to prevent undefined behavior. Most interpreted languages, like Perl, Python, and Java, are safe languages.

For example, in C#, the boundaries of arrays are checked at runtime, so that running off the end of an array has the predictable effect of throwing an `IndexOutOfRangeException`:

```
/* This is C-sharp code */
a = new int[10];
for(int i=0;i<100;i++) a[i] = i;
```

In a **statically typed language**, all typechecking is performed at compile-time, long before the program runs. This means that the program can be translated into basic machine code without retaining any of the type information, because all operations have been checked and determined to be safe. This yields the most high performance code, but it does eliminate some kinds of convenient programming idioms.

Static typing is often used to distinguish between integer and floating point operations. While operations like addition and multiplication are usually represented by the same symbols in the source language, they are implemented with fundamentally different machine code. For example, in the C language on X86 machines, `(a+b)` would be translated to an `ADDL` instruction for integers, but an `FADD` instruction for floating point values. To know which instruction to apply, we must first determine the type of `a` and `b` and deduce the intended meaning of `+`.

In a **dynamically typed language**, type information is available at runtime and stored in memory alongside the data that it describes. As the program executes, the safety of each operation is checked by comparing the types of each operand. If types are observed to be incompatible, then the program must halt with a runtime type error. This also allows for