Hash Table Implementation

The symbol table was designed around the idea of an struct called ‘hash’ that would hold all information about the hash. This included the current lexical level of the program, a pointer to the hash’s hashing functions, and an array of struct ‘hashElement’ pointers. We created an global variable called ‘symbolTable’ that created a version of the hash struct.

```
struct hash {
	struct hashElement *elements[TABLE_SIZE];
	unsigned int (*hashFunction)();
	int lexLevel;
};
```

We decided to include the lexical level within the hash table struct because it allowed us an global access at all times to it. Additionally, it made sense to keep the program lexical level counter with the symbol table because of the close relationship between them i.e. the ability to pop and push lexical levels as symbols were added to the symbol table.

The decision to include a pointer to a hashing function came about for a variety of reasons. First, we wanted the ability to change hashing functions at start up. The primary reason for this was for testing purposes. With a good hashing function it can be very hard to find collisions. For this reason, in many of our unit tests we used a simplified function that was guaranteed to have collisions. This way we could properly test all the functionality of the symbol table. The second reason, was because we wanted to keep the all the hash components together. This way, there was only one point of access for everything dealing with the symbol table.

The decision to create an array of pointer hashElements was very intuitive. By having a hashing function take a string key and generate a number, we were able to use the resulting value as an index to the array. This ensured an efficient lookup time.

```
struct hashElement {
	char *key;
	struct Symbol *symbol;
	struct hashElement *prev;
	struct hashElement *next;
};
```

Each member of the array was a pointer to a hashElement struct. We made these structs so they could be used to create an doubly linked list. This ensured a bucket like structure for hash collisions. When we encountered a collision, we attached an element to the bucket linked list. By allowing the linked list to be doubly linked we also increased efficiency but also ease of use.

Each hashElement also include a pointer to a symbol struct. This was where the hash really started to become a symbol table. Each symbol struct held everything we needed to identity elements.

The hash, while allowing collisions did not allow key duplicates at the same lexical level. To this end the creation function would report errors and issues to the callee. A interesting feature of the hash was that when a null key was passed, we would create an unique key for it by querying the system time. This was particular useful for anonymous types as it insured no collisons.

Symbol table Implementation

Symbols

A Symbol was represented in our program in the following manner (based largely on the Piotr Rudnicki's notes available at **URL**.):

```
typedef struct symbol {
        char *name;
        kind_t kind;
        Kind kindPtr;
        int lvl;
        Symbol *next;
} Symbol;
```

-
- The 'name' field was used as the both the identifier of the Symbol throughout the program body and the key used for lookup in the hash table.
-
- The 'kind' field was an enum type giving indication as to the nature of the information stored with in the rest of the Symbol body. Together with the associated union kindPtr, we were able to associate with each Symbol a unique set of fields which held the information which would be required in order to insure the semantic correctness of program our compiler was to parse. Each Symbol could be one from amongst the following Kinds:
-
    -
    - CONST_KIND : in this case the Symbol represented a constant value defined in the const declaration section of the program head or a procedure/function body. Such a Symbol had an immutable value associated with it at the time at which it was parsed by our compiler – the value of the constant was of course dependent on the type being assigned.
    -

    - FUNC_KIND : in this case the Symbol represented a declared function within the program. The associated list of parameters which the function expected to receive on invocation as well as the type returned by the function.This was done in order to insure that we were able to confirm that a function was invoked with the correct number of parameters of the correct type
    - PARAM_KIND : Symbols of this kind represented parameters used to invoke functions. They were treated as Symbols although this was only done for the convenience of their representation in this format.
    - PROC_KIND: in this case the Symbol represented a procedure declared within the program. The list of parameters which the function expected to receive upon being invoked were kept in the kindPtr in order to insure that they could be added as local variables to the lexical level appearing immediately after a procedure/function declaration and were in a convenient data format for checking the correctness of function invocations in the program.
    - TYPE_KIND : Symbols of this type had their associated kindPtr point to a structure which was a type specific definition of the type that the symbol was meant to define. Types and what information structure each Symbol had is detailed in more depth at a later time in subsequent section.
    - VAR_KIND : Symbols of this type were used to represent the variables in our program. Their associated kindPtrs held only the type used to define the variable such that semantic correctness in uses of the variable in expressions and assignment operations could be proven to be correct.

- The integer value level was used to keep track of the lexical level at which the Symbol had been defined.

The pointer to a next Symbol was used to both to link Symbols at each lexical level in the hash table as well as to link Symbols together while attempting to build a structure element from the grammar (e.g, in the indexing of multiple dimensional array by comma separated values).

The decision to use the above given structure for a Symbol was not an easy decision and many alternatives were considered, however in the interest of keeping a clean mental model of what a Symbol represented (and to keep to deadlines) the above given model was used though it lead to less than desirable levels of indirection in our code and would have been revised had we the time.

Types

For Symbols of kind TYPE_KIND, we associated a type specific structure which gave the information that needed to be associated with a particular type in order to describe it adequately for the purposes of semantic analysis.

- Bool, Char, Int, Real, all stored obvious associated values.
- String type structures stored both a string and the length of the string associated with a particular instance of the string type.

- Array type structures stored pointers to the type symbols which defined the types used to index the array and the type of each entry associated with the array.
- Scalar types stored a list of the Symbols of the integers constants over which they iterated.
- Subrange types structures stored the Symbol which defined the type which their indices take as well as the low and high values that defined their range.
- Record types had their own private Symbol table for which to store their field entries -- this was done largely for the ease of its implementation and the avoidance of distinguishing

Grammar Changes and Action Execution

In order to properly propagate desired values and structures to places in the grammar in which they would be usefully purposed changes to the grammar used in the first part of the compiler had to be made.

In particular, difficulties with regard to chaining function and procedure invocation and declaration parameters in order to construct lists of Symbols which could be associated with the given procedure/function

Moreover, the the requirement to support both indexing arrays by comma separated values and blocks of square brackets presented challenges as each method of index the array constructed the associated list of indices in different ways although both had to ultimately resolve to a structure which provided adequate information to insure the correctness of an array indexing.

Error Reporting Strategies

Error reporting was done as close to the source of the error as possible in order to be able to provide to the user the most context in which the error occurred. Moreover, once an error had occurred and propragated to the top level of the grammar there was little that could be done in terms of attempting to provide a best-effort, coherent semantic model of was was actually intended by the programmer in their incorrect code.