Matthew Low, Anthony Galati, Stevan Clement, Mike Bujold

The man page can be viewed by navigating to the submission folder and typing the following:

man documentation/pal.1

HANDLING OF LEXICAL UNITS

Our lex program parses through tokens in the following order: first we handle comments, both single and multi-line. Multi-line comments are recognized in two parts. We first acknowledge a multi-line comment by the first brace, but in order to continue counting newlines for our error reporting we view the closing brace as a separate token.

We then parse reserve words. Initially we'd included 'true' and 'false' in our reserve words, but later realized this was an error. We now assign them default values in our symbol table, which allows them to be overwritten, as is allowed by PAL.

Then we parse our real consts, strings, unterminated strings, IDs, and integer consts. This is followed by our other recognized symbols, and garbage collection. Garbage collection occurs silently, with lex calling an error function which then notifies the user.

SYNTAX ERROR REPORTING STRATEDGIES

Syntactic errors were detected by placing the 'error' token in reasonable positions. This was generally at high level positions, such as at the program head and around const/type declarations. We have also toyed with using the token in lower level but more high risk areas, such as near variable assignment with mixed results. This often introduces new shift/reduce conflicts, which is less preferable than simply trashing more input to allow for recovery. The yyerror function then uses the information we store holding the line and column position to provide a reasonable message to the user.

Since line endings have little meaning in the PAL language, we don't use the yerrok macro, as there is no way to match an error to a whole line (to do so may mean an error matches the whole program). As such, if we encounter an error, we don't want to output a bunch of other errors that are immediately caused by the first.

RESOLVING CONFLICTS IN BISON GRAMMAR

The conflicts in the bison grammar were mainly the result of ambiguities created by multiple similar paths to the grammar's atoms. The most serious example of this was how both "type" and "expr", two similar rules, reduced along separate paths to simple_type. We were able to remove this ambiguity from the grammar by removing the rule for reducing to simple_type from those available to expr, and instead replacing it with the applicable atoms.

Another issue with the grammar was the possibility of declaring an array with type anonymous enum. We replaced this rule with two new rules, one of which allows for instantiation of strings and one which allows only previously declared variables as a type.

IMPLEMENTATION OF THE SYMBOL TABLE

The symbol table entries follow the class notes in their design. The symbol table itself is an array of linked lists of symbol table (sym_rec) entries. The array is size 17 to correspond to the display register values in ASC and allows for a base level of predefined values which we have hard coded into the compiler.

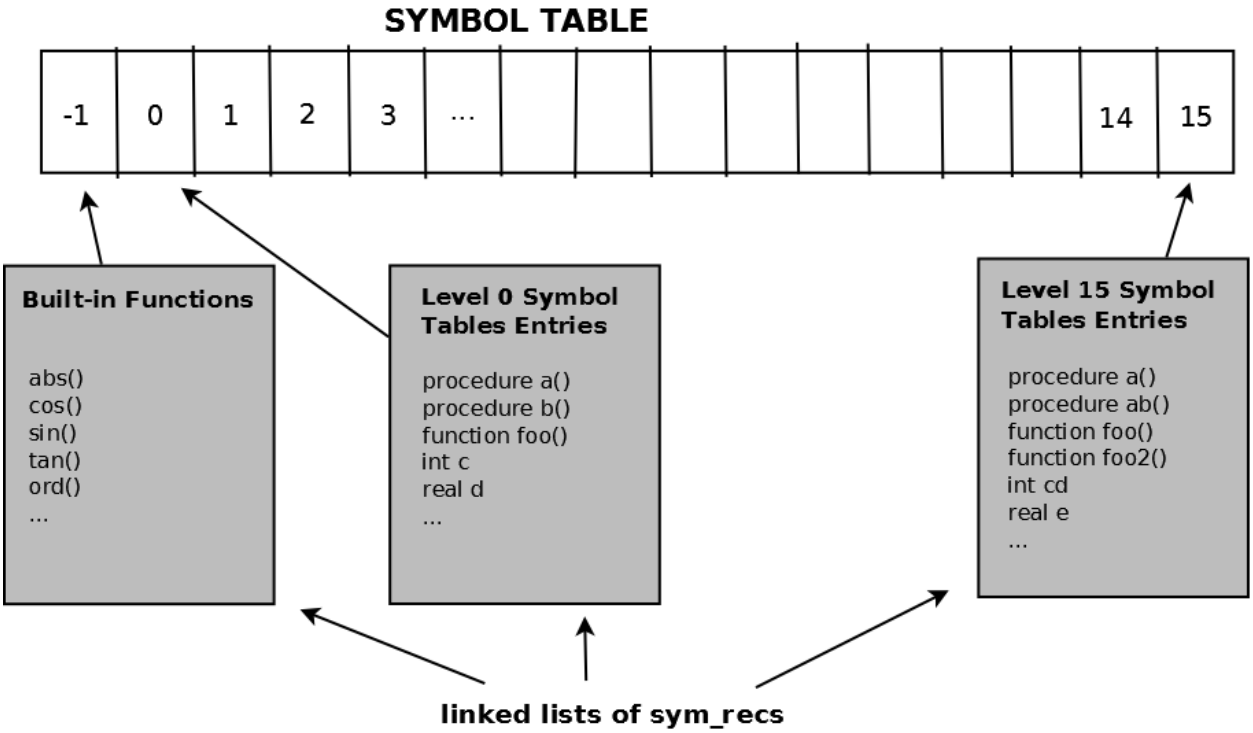

Fig 1. *This is a basic overview of our symbol table. The base level contains all of the built-in functions the compiler offers. Each of the subsequent levels is employed when a nested call is made, up to a maximum nesting level of 16.*

The symbol table includes a few basic functions as an interface to it. PushLevel and PopLevel are included to change the scope of the program - these are called in Bison when we begin to parse inside a

new procedure or function. If we go over the allowed number of levels the compiler will exit gracefully since we can't go any further with the display registers. Various functions have been created to add items to the symbol table - one for each of the main types (var, const, type, func, proc). We have included a symbol table printing procedure for debugging purposes.

The symbol table consists of sym_rec structures, which are at the main level of the symbol table. These structures have a class number (eg. OC_TYPE) which defines a field in a union containing more class specific information. The most interesting of these is the struct type_desc including the types of allowed types. This field is important to comparison later on, as we are able to pull out of it the address of a pointer that was created when a type as declared to compare it with other types. Every variable and constant is linked to a type, so through many levels of redirection we are able to get all of the information we require about an item in the symbol table.

We initialize the table with all of our built in functions, as well as values for true and false. The built-ins reside at the bottom level (-1) of the table.

SEMANTIC ERROR CHECKING AND SYMBOL TABLE USAGE

All semantic error checking is handled inside of the Bison grammar - there are a few helper functions in semantics.c and symtab.c, but the Flex code does nothing other than return tokens. Because of this, we were able to remove INT, REAL, BOOL, and CHAR from the bison grammar file, allowing Flex to pass Bison just the ID token with the identifier name that it parsed. All integer constants are handled with INT_CONST and real constants with REAL. In the grammar rules for declaration, most variables are handled by receiving an ID token and doing a local lookup with the string that was passed with it.

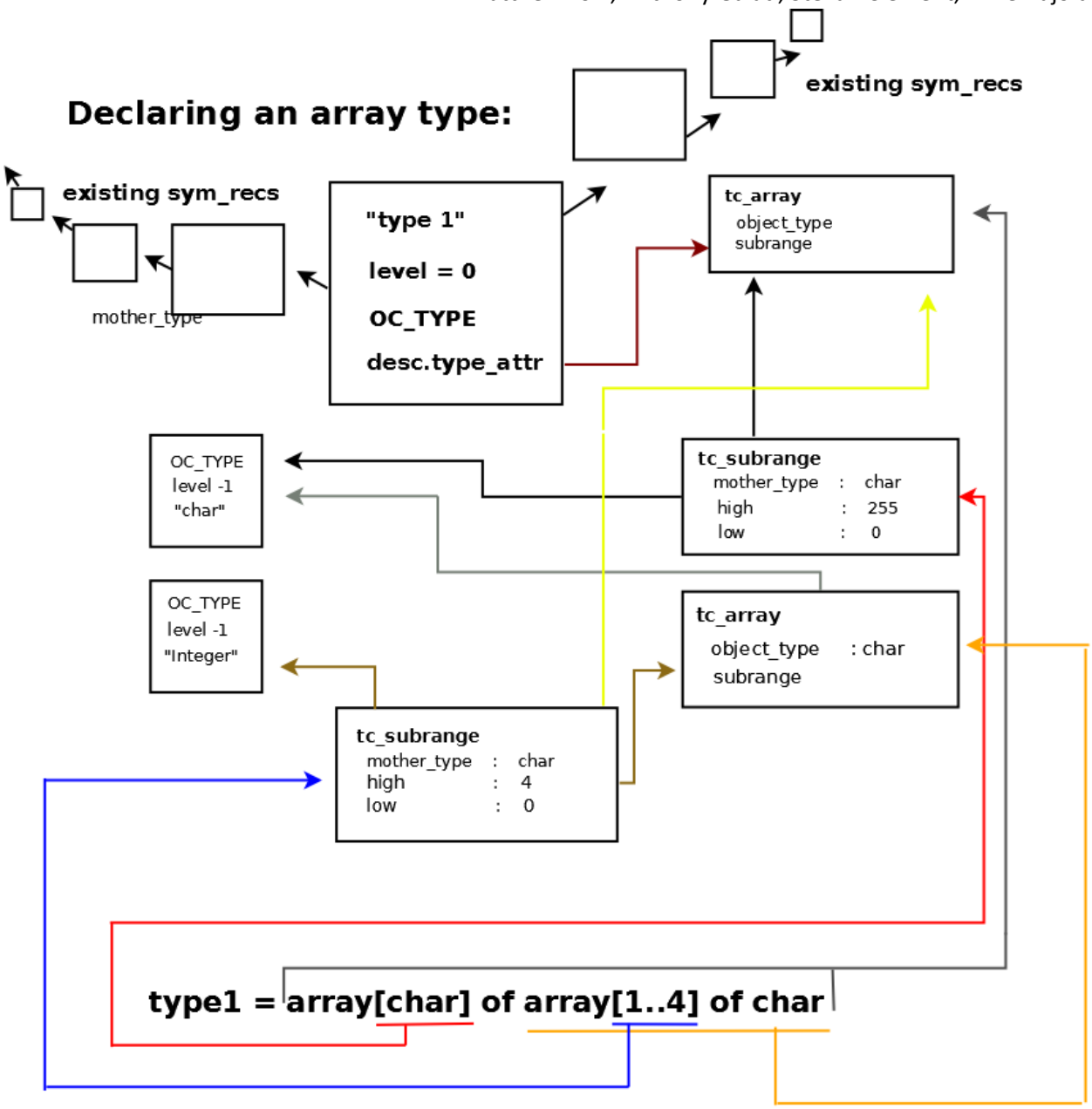

Fig 2. *This is a symbolic depiction of the logic used in our implementation of the symbol table. In this case to declare a type composed of nested arrays.*

Populating the symbol table is straight forward for most classes. During function or procedure parsing the expected parameters are added to the next level of the symbol table as well as copied over to a linked list of expected parameters attached to the entry for the function or procedure at the current level. While parsing the parameters, if it is found that one of them causes an error (if it has already been declared or is of an invalid type), we put it in the parameter lists associated with the function or procedure and give it type OC_ERROR. Enumerated types are handled somewhat differently. As the grammar parses them they are collected in a linked list and sent back through the grammar using $$

until a type needs to be defined or a variable needs a list of associated scalar types. Consider the following expression:

```
begin
   var room : (chair, table, door);
end
```

In this case, the scalar list would be parsed, and when it was finally realized that we were assigning a variable to a type of this list, a new anonymous type would be created with no name, only available by checking the type of the variable.

Type checking is currently done for expressions by simply grabbing the type of whatever constant or variable we have seen and passing it through the $$ variable. If an error is seen at any point in an expression evaluation, an error is reported and $$ is set to NULL. We decided to do this because sending an arbitrary type through an expression would cause more errors than it would detect.

To check functions and procedures, since Bison sees the name before the parameter list, we save the important parts about the function in the struct plist_t and pass it through $$ as we parse the parameters. This way we can know about missing parameters and incorrect ones, and give functions a proper return type.

ASC CODE GENERATION

The ASC code generation is performed by using a series of emit functions housed in our pal.c file. There are emits which handled each of the of the ASC instructions (PUSH, PUSHI, POP), many of the most common cases (such as adding two integers or reals), as well as a few more specialized cases such as run-time array bounds checking.

Less trivial cases are written out in ASC in our function libraries. These libraries include code to handle cases such as manipulating strings, arrays, comparing values, etc. The compiler links these libraries for run-time use, at compile time. Interestingly enough in one instance as we attempted to implement the arctan built-in we used the compilers basic emit system to write more complex code.

CHALLENGES AND SOLUTIONS

Our first challenge coming into the project was altering the grammar. As previously mentioned there were ambiguities that allowed for multiple paths to the rules for simple_type. We were able to remove these ambiguities by shifting the atoms found in simple types upwards, allowing for higher level rules to account for the removal of the ambiguous pathway.

We also had issues with multi-line comments. Specifically in that our line counter would not be updated for their span. We solved this problem by breaking multi-line comments into two tokens. This allowed newline to be returned while the comments were eaten up, making our line reporting more accurate.

Error reporting and recovery were both big challenges. Problems with cascading errors, especially those associated with missing semicolons gave us a lot of trouble. To combat this problem we watch for instances when we know we are expecting a semicolon, and subtracting from our line count to produce more accurate error reporting.

Implementing the symbol table was probably the most difficult portion of the project. There was a great deal of trial and error associated with getting type checking down, and again implementing useful error reporting. There was no single fix for these issues. It was a matter of testing and iterating on the code. This was where we spent the majority of our man hours.

Working towards ASC code generation contained a number of issues. These were often centred around how a design decision for a portion of the compiler, which was previously implemented, now hampered our ability to implement some other aspect. In such cases we were forced to revisit parts of the compiler repeatedly which was very time consuming. We quite often ran into this kind of trouble when tackling various aspects of string/array manipulation. Another problem we encountered was when implementing the reading functions we were taking in a newline character when return was pressed. We resolved this eating the final character regardless of what it is. In the Pascal standard strings are space padded, so we simply treat the end of a string as padding regardless of whether it is a space or newline character.

REFERENCES

"Bison - GNU Parser Generator." The GNU Operating System. Free Software Foundation (FSF).

Web. 04 Dec. 2011. <http://www.gnu.org/software/bison/manual/>.

Donnelly, Charles, and Richard Stallman. "Bison, The YACC-compatible Parser Generator."

The LEX & YACC Page. Web. 04 Dec. 2011. <http://dinosaur.compilertools.net/bison/index.html>.

Paxson, Vern. "Flex - a Scanner Generator." The LEX & YACC Page.

Web. 04 Dec. 2011. <http://dinosaur.compilertools.net/flex/index.html>.

Levine, John. "Flex & Bison." Google Books. O'Reilly Media, Inc.

Matthew Low, Anthony Galati, Stevan Clement, Mike Bujold
Web. 04 Dec. 2011. <http://books.google.ca/books?id=3Sr1V5J9_qMC>.

Rudnicki, Piotr. "CMPUT 415 Class Notes." In-Class. First accessed 09 Sept. 2011.