CMPUT 415 - Team YACC - Compiler Documentation

Sarah Anderson
Marko Babic
Guanqui Huang
Aaron Krebs
Shihao Xu

Further Improvements on Semantic Checking

Since checkpoint 2, we have made further improvements on semantic error checking. If a program re-declares a type, we show an error and do not perform the re-declaration. if the user re-declares a type, there are two likely scenarios for how this type will be used in the remainder of the program we are parsing. The first scenario is that the user intends the first type declaration and uses the type as though the first declaration were valid, in which case if we assume the second declaration overrides the first we would show many semantic errors incorrectly. Alternatively the user intends the second type declaration to be valid, and if we assume the first is valid we would also show many incorrect semantic errors.
This leaves us with two equivalent scenarios where we could choose to use the opposite type declaration from the user and report many type-related semantic errors incorrectly. We have chosen to use the first declaration of the type when parsing the remainder of the program.

Semantic vs Syntax Errors

In a few cases we could choose whether to catch errors at the semantic level or syntax level. Catching them at the syntax level would involve re-writing parts of the grammar (which is why there was a decision to be made in the first place). We usually chose to catch these errors at the semantic level since this involves fewer re-writes of the grammar.
One such decision was whether to catch array invocations involving reals and function calls at the syntax level (for example: ‘a = array[1..2E12] of integer’) or not. Catching this at the syntax level would involve duplicating large chunks of the grammar (the entire expr tree), which we didn’t want to do. Thus, we have opted to catch this type of error as a semantic error.

Calling the ASC Interpreter

On a successful parse, we need to call the ASC interpreter. To do this, we have opted to spawn a child process using fork() and then wait for the child to return. The child process, in turn, uses execl() to replace run the ASC interpreter. If the ASC interpreter returns with an exit status, out child process does as well. This way, we can report error conditions in the invocation of the ASC interpreter gracefully.

The Chapter in Which we Finally Use an Expression Tree

In attempting to emit code for this deliverable, we realized (unfortunately only two days before the deadline) that a syntax tree is necessary in order to generate code in a sane way. Although for a long time we used various work-arounds to generate code for a lot of things, using a syntax tree really removed a lot of the corner cases. The greatest difficulty lied in salvaging old code such that it could be used in the new design. Of particular annoyance was the restructuring of the grammar and semantic analysis functions around the fact that we were no longer dealing primarily with passing around symbols in the grammar but rather expression nodes. Formulating walks of the tree in such a way that the emission code formed on the basis of the non-existence of a syntax tree produced awkward code -- requiring us in several places to walk the an expression node to produce an expression then push individual values to the stack afterwards that were not included in expressions due to the structure of the old code. In retrospect, this was our biggest design mistake: it should have been the case that when we were uncertain with regard to the need of a syntax tree that we mapped out how expressions were going to be emitted without one. Once we arrived at function invocations, we would have likely realized that the need to maintain expressions components such that they can be pushed onto the stack as needed is critical in order to properly handle situations in which we do not know if the value an expression is ultimately to resolve to is a value (in the case of regular parameters to a functions) or an address (in the case of passing parameters by reference).

This is a One-Pass Compiler

Our implementation is a one-pass compiler. We are able to do all syntax validation, semantic checking, and code generation in a single read of the file (emitting the code listing takes a second pass, but that doesn't count, right?). Specifically, we append errors to a linked list as we go, and append emitted code to another linked list. At the end of the parse, if we found no errors, we emit the link-list of ASC statements to the .asc file, and write the original input file, interspersed with errors, to the .lst file.

What Works - Implementation Details

We are confident in our ability to find syntax and semantic errors, as well as to generate code in most situations. Apart from the bus listed below, we are not aware of any missing functionality. For constant values, our compiler is able to pre-evaluate simple expressions depending on different operator tokens at semantic level. If both sides of operator are constant and are type compatible, the type of the result symbol will be corresponding to one of the type of these constant symbols, if they are not, then the compiler will return a type symbol which points to null value and reports an error.

Known Bugs and Problems

We occasionally fail to insert symbols into the symbol table. The compiler exits gracefully in this case, but parsing/compilation cannot continue even though the given PAL may be valid. We know the error is not due to hash key collisions (those are unit tested extensively), but we did not have time to track it down further. The error has occurred less than a dozen times, but has been present since the initial hash table implementation.

When a string contains invalid characters (such as \a or \q, for example), we fail to recognize the start of the string properly. Thus, in most cases, we see the end ‘ of the string as the start of a multiline string literals. Thus, the string literals end up being mismatched, and the parser will continue to consume more characters until it hits a start of the next string literal. Once this has happened, there is no way for our compiler to recover and correctly checking the rest of pal program.

Our compiler does not support recursion. This is, unfortunately, a result of how we handle scope for functions. We make the function name available as a variable (to handle return values), but do not handle that the function name should also be in scope as a callable function. We noticed this too close to the deadline in order to correct it, but a good approach would be to treat the function name as a special case that needs to be checked whether it is used in an assignment or a function call.

We leak memory. Everywhere. Primarily, this is because we do not free memory associated with symbol table symbols when we decrease the lex level. Although we remove the symbols from the symbol table, we do not free the associated memory. This is largely due to our chosen symbol architecture. It is quite convoluted with many different structs, pointers, and levels of indirection. We did not want to free memory because in many cases, many symbols hold pointers to other symbols at a lower lexical level in the symbol table. In these situations we were not confident when a pointer could be freed safely. In hindsight, a simpler design would simplify this, and would likely simplify many other parts of our compiler.

Extra Features

All additional features are documented in the man page. We added the suggested ‘-c’ argument to suppress invocation of the ASC interpreter.We also print all error reporting to standard out in addition to producing the listing file. This can be disabled with the ‘-q’ option.

Implementation Defined Constants and Types

As per ANSI-ISO 7185, the constant maxint is implementation-defined. We have chosen to give maxint the value of 2147483647, as this is the MAX_INT defined in the ASC source code (which makes sense, since it’s the upper limit on a signed int in c). We check that integer constants are within the allowable range (-maxint to maxint) when as we parse. For reals, our implementation handles them as double type internally, and we rely on the error checking of strtod() to make sure the values are valid.
Note also that as per section 6.4.2.2 (Required simple-types), the pre-defined types integer, char, and real are implemented. The Boolean type is implemented as well (note the uppercase spelling).

Generating Labels

From Jonathan’s notes, we saw to build a stack of labels for use with IF statements. This stack turned out to be quite handy, as we ended up using it in multiple places. IF statements, WHILE loops, procedure names, and labels for procedure main bodies all use a unique label stack to keep track of things that are (independently) nested.