PAL Compiler Documentation

Chris Pavlicek, Connor Moreside, Mike Armstrong, and Steve Jahns

December 6, 2013

Overview

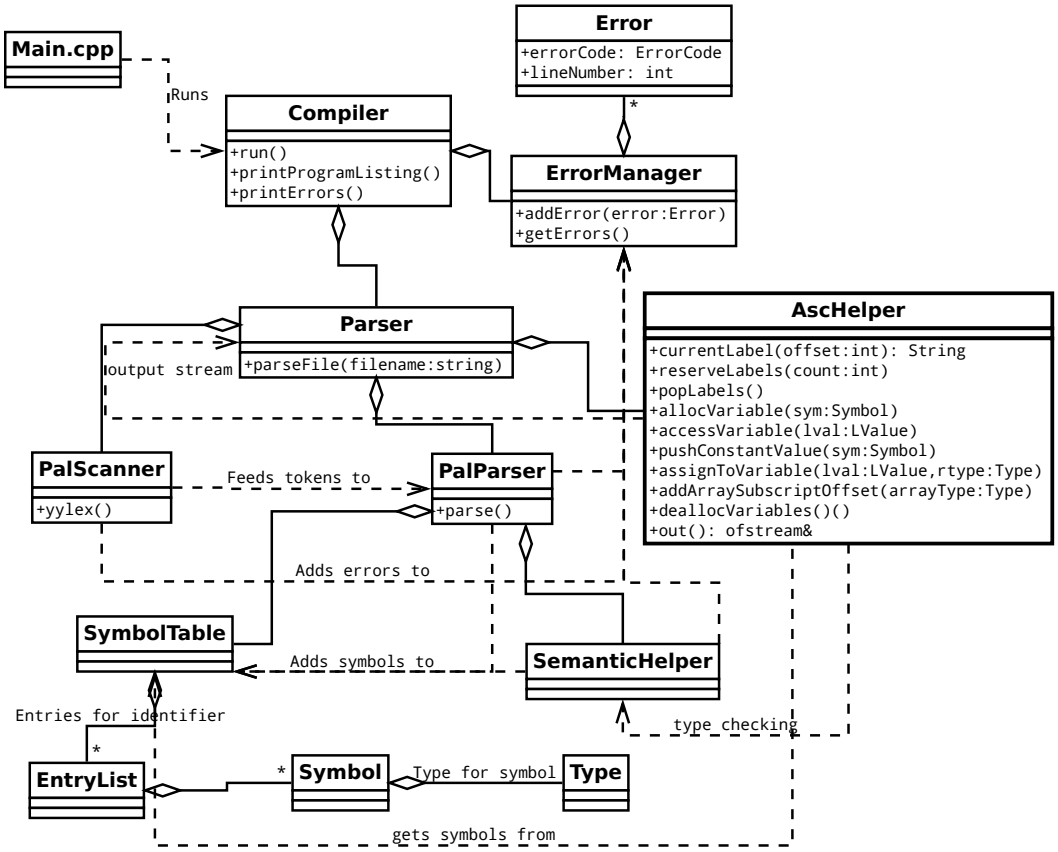

Figure 1: General structure of the compiler's parsing and semantic analysis components.

At this stage, our compiler source consists of 10 classes. The `Compiler` class is instantiated by the program's `main()` function, and ran with the program's given command line arguments.

The `Compiler` class creates an `ErrorManager` class which tracks errors as they are detected while parsing the file. It passes the `ErrorManager` to the `Parser` class when it is constructed.

The `Parser` class is a wrapper for the `flex` and `bison` generated `PalScanner` and `PalParser` classes, from the `pal.lex` and `pal.y` sources, respectively. `Parser` parses a file with `parseFile(fileName)`, which creates a `PalScanner` and `PalParser` where the scanner runs through the file and feeds the parser with a stream of tokens. The scanner and parser are given access to the common `ErrorManager`, and when they

encounter errors during lexical and syntactical analysis, they create new `Error` objects with the appropriate error code and line number and give them to the `ErrorManager`.

Semantic checking is primarily completed by the `SemanticHelper` class and of course the symbol table (`SymbolTable`). We currently have a monolithic symbol class (`Symbol`), which can represent the various identifier constructs available in PAL.

When parsing and semantic checking have finished, the `Compiler` takes the errors accumulated by the `ErrorManager` and either displays them inline with the program listing, or all errors in order of appearance without the program listing if `pal` was invoked with the `-n` option.

Lexical Analysis

Our lexical analyser is generated by GNU Flex from the source in `./src/pal.lex`, which is bundled into the `PalScanner` class. It is capable of recognizing the following errors at the lexical level:

Unclosed Comment If a comment is started with { but is not followed by a } later in the file, the scanner will report an error. In our scanner, the comment rule will match to the end of the file in this situation, so unfortunately no more analysis of the file is possible at this point.

Unexpected Comment End If the scanner encounters a } when it is not in a comment, it will report an error. Otherwise, the extra brace is ignored and analysis continues in an attempt to find more errors in the file.

Multi-line String Literal In the PAL language, a string literal cannot span multiple lines, so if the scanner encounters a string literal that extends beyond the next newline, it will report an error that indicates the start and finish line of the illegal multi-line string. It will still return a `STRING_LITERAL` token, and continue analysis to try and find more errors.

Invalid Character or Escaped Character in String The scanner will report an invalid character/escaped character in a string literal if it contains either of the following:

- a \ that is not followed by a `'`, `n`, or `t`
- a \ at the end of the string, that is not preceded by another \

Invalid Identifier Valid PAL identifiers start with at least one letter, followed by any number of letters or numbers. Identifiers that start with numbers will be reported as an error.

Invalid character If the scanner detects an unknown character, such as `!` or `#`, it returns an error indicating that an unrecognized character was found. It easily recovers from this error and continues on.

Syntax Analysis

Our syntax analyzer is generated by GNU Bison from the source in `./src/pal.ly`, which is bundled into the `PalScanner` class. It is capable of recognizing the following errors at the syntax level:

Program Declarations The compiler detects faults in the declaration. If it is missing a program will not compile. Errors will be outputted for the following cases:

- If the semicolon is missing
- If one or both arguments are missing
- If the program is missing a closing parentheses, starting, or both.
- If there is an invalid program name

Variable, Constant or Type Declarations If a variable, constant or type is incorrectly declared, the compiler will catch the following errors:

- If an improper type of assignment is used. IE. constant is declared using :, instead of = an appropriate error message will be emitted for each type.
- Improper variable names, only variables starting with a letter, and containing letters and numbers after are valid.

Enumerations If an enumeration is incorrectly defined an error will be produced.

Arrays Arrays which are declared incorrectly, either by defining the set wrong, or by specifying an invalid type will produce a specific error.

Functions Functions will be checked for proper formatting if a function is invalid an appropriate error will be emitted. Functions are required to have return types, enclosing parameters and a ending semicolon. Each function is required to have variables declared before the begin statement and required to be have a begin and end statement.

Procedures Procedures are declared in a similar fashion as functions, and have the same error checking except that they don't have a return value and are ended by a period instead of a semicolon.

Expressions Expressions will checked for proper structure. If a symbol is missing, semicolon or any part it will be discarded and return an error.

Recovery Strategies Try to catch errors as early as possible by defining strict cases that are simple to implement but are common user errors. Errors which are not defined are given a generic error output, using the built in error within bison.

Semantic Analysis

Semantic analysis first begins within the bison grammar file, `./src/pal.y`.

There exists two main classes for encapsulating information regarding the type of a particular symbol, and the symbol itself.

Firstly, we have the monolithic `Symbol` class, which is responsible for capturing important information regarding any given identifier in a PAL program, including variable declarations, function declarations, enums, and so forth.

And secondly, there is a class which encapsulates information regarding types in PAL, which was aptly named `Type`. This class is responsible for holding information about any type that can be declared in a valid PAL program. It has fields for records, functions, primitive types...

We make use of a class called `SemanticHelper` to encapsulate some of the semantic analysis functionality in order to keep the grammar file relatively clean.

The `SemanticHelper` performs the following functions:

Pre-Populate Symbol Table There exists a function which populates the symbol table with various pre-defined types, functions, constants, etc...

Perform Type Comparisons Since the `SemanticHelper` class has a reference to the symbol table, it is able to perform a type comparison between two separate constructs, such as variables, constants, etc...

Add New Types To Symbol Table When a new type is encountered in a PAL program, the `SemanticHelper` is responsible for creating a new `Symbol` and corresponding `Type`. This information is then added into the symbol table.

Retrieve Type After Expression Evaluation During the evaluation of certain statements, the final type of the an operations must be determined in order to determine type compatibility. The `SemanticHelper` is responsible for this.

Retrieve Primitive Types Pre-defined ŕawt́ypes such as boolean, integer, real, etc... are retrieved using this class.

The symbol table is contained in the `SymbolTable` class. Our symbol table was built around the C++ predefined `unordered_map`. This was quite nice since we did not have to define our own hash map implementation, thus greatly reducing the chances of bugs due to improper implementation.

Within a bucket of the top level hash map, there is a class called `EntryList`. It is essentially a vector containing the definitions of symbols at various lexical levels. The symbol table's methods handle the different definitions at various levels with ease.

The Symbol table contains the following functionality:

Adding Symbol

Finding The Definition Of A Symbol

Retrieving A Symbol

Retrieving The Current Lexical Level

Incrementing The Lexical Level

Decrementing The Lexical Level

As you can see, the `SymbolTable` and `SemanticHelper` classes work very closely together in order to perform semantic checking.

Our Failed Abstract Syntax Tree

When beginning Checkpoint 2, we started to create an AST structure so we could cleanly separate semantic analysis and code generation functionality from the grammar file. Initially, it seemed like a great idea, using proper software engineering principles and such. As time progressed, the amount of code began to spiral out of control, leading to a convoluted mess of AST nodes. The amount of code need to properly build and traverse the AST was preposterous. We decided to start from scratch essentially. Let's just say after we reverted back to our checkpoint 1 codebase, there was over 9 000 lines deleted! Lesson learned...

Code Generation

The code generation portion of our compiler primarily lies within a class called `AscHelper`. This class has numerous responsibilities such as type conversion, allocating variables, logical expression, etc... It is also responsible for generating ASC code to call the built-in routines. AscHelper will stop generating code after the first error is encountered; this was done to speed up the compiler, as well as to ensure that no unitialized or improperly set values are used.

`AscHelper` contains a reference to our `SemanticHelper` class as well in order to determine proper code generation for integer, character, and real based types.

We chose not utilize an AST for code generation; mainly because we would not be doing any sort of code optimization. This design choice was reinforced by our previous failed attempt at creating a manageable AST in the previous checkpoint. Furthermore, we believed that building and traversing an AST would be slow and prone to bugs.

Code-gen details We decided to inline the code generation with the parser. This means that the code is generated while the parser is parsing the file, rather than after the parser is finished. Thus, our compiler is a one-pass compiler.

- `Allocation of space` - We allocated space using the ADJUST instruction, at the time of variable declaration, based on the size of the type of the variable.
- `Expression evaluation` - This was very simple to do, as Asc is a stack based machine. In essence, most of the basic operations simply needed to have their Asc equivalent pushed into the code stream, as it was assumed that the two operands would be in the correct order on the top of the stack.

- `Variable references` - This was done using the symbol table. A PUSH instruction was used, in conjunction with the symbol table information, to push the variables value onto the stack.
- `Variable assignment` - This was done in a similar way to reference, but in reverse using a POP instruction.
- `If-else statement` - This was done very similarly to the class notes.
- `While statement` - This was also similar to the class notes.
- `Exit-Continue statements` - This was done using GOTO statement in Asc, and a While statement stack within the compiler. Each encountered While loop would have it's start and end label pushed onto the internal stack. Then, if exit or continue were seen, a GOTO would need to be pushed into the code stream with the correct label (continue=start, exit=end). Upon leaving a while loop, the internal stack would pop the top labels off of the stack.
- `Function calls` - To when calling functions, we used a calling convention. The caller is repsonsible for pushing the arguments onto the stack, in left to right order, as well as making space for the return value after the arguments have been pushed onto the stack. Then, the caller calls the function, using the lexical level of the function as a display register. After the callee has return, the caller then can use the return value and remove the space for the arguments on the stack.

Built-in Routine Library All of our routines were written directly in ASC and then later connected to predefined PAL function symbols using the `SemanticHelper` class. Here are descriptions of the non-trivial routines:

- `sin(x)` - Approximation using a quadratic curve
- `exp(x)` - Approximation using a Taylor Series
- `sqrt(x)` - Utilizes the Bakhshali Approximation

Note that due to the extreme inaccuracy in floating point numbers within the Asc interpreter, some builtin functions will be relatively inaccurate.

Runtime Error Handling We took a very simple approach to runtime error handling. In most cases, an error message is printed out and the ASC operation `STOP` is called. No attempts to recover are made in most cases. Some of the runtime errors that we handle in our compiled code:

- `Bounds checks` - We check to make sure that all accesses are within the specified range for the array type. This is only true when the bounds checking is not disabled (-a). If it is disabled, we do not do checks, and the code should run faster for array accesses.
- `Division by Zero` - We handle division by zero in our generated code, both at compile time and runtime. At compile time, constant expressions are evaluated, and if they contain division by zero, and error is shown and the code is not (and CANNOT) be compiled. At runtime, if a division by zero occurs, an error message will be shown and the code will exit.
- `Invalid arguments to builtin procedures` - We check to make sure that the argument passed to the builtin is valid. If it is not (in the case of sqrt(-1), etc...), we throw and error and the program will exit. This occurs at runtime.

Testing

GTest For accurate tests of the compiler we used gtest. Within gtest we built specific tests using tokens. Our tests can be ran using, `%make test`, `%make ScannerTest`, or `%make ParserTest`. (make test runs every test). Over time, we managed to gather a huge amount (over 500) of test files, and this made it dramatically easier to make changes to the compiler, as it was very easy to see when a change had broken something.

- `MockScanner.cpp` is a simple file. It defines a " MockScanner ", for use with testing. The Parser can be unit tested without the dependence of the real scanner which may contain errors.

- `ParserTest.cpp` contains a few tests which push tokens onto a stack and are fed to our parser to check if the grammar contained in `pal.y` is recognized correctly.
- `ScannerTest.cpp` Is used to check pal programs. By hand expected tokens are placed into this file, and if they match from the input file the test will pass.
- `ParserTestWithFiles.cpp` This file is automatically generated using a bash script. This file takes the tests in the test_cases folder and builds a cpp file, if it is labeled vt*.pal the expected return is 0 (valid pal) or bt*.pal is 1 (invalid pal). Full test naming conventions can be found in NAMING-CONVENTIONS.txt, within the test case directory. This made it very easy to add test cases and to run them while making changes to the grammar or token identifiers.
- `CodegenTest.cpp` This file contains a large amount of valid test cases, all of which compile to Asc code. Each of the test cases compile and run the specified code, and then check for the expected output. If the output is the same, then the test passes, otherwise it fails. There are approximately 130 test cases within this file.

Submitted Tests 0.pal - 9.pal Tests 0-3 contain only semantic errors; a maximum of 3 errors per test file. Tests 4-7 comply to the PAL specification and therefore compile properly, producing ASC code. Test 8 checks various built-in routines to ensure that they are functioning as expected. Test 9 (THE TEST OF DOOOOOOOOOOOOOM!!!!!!!!!!!!!!!!) is a fairly convoluted sequence of nested recursive functions, multi-dimensional arrays, and crazy nested while loops, which is, surprisingly, correct and should output 42. These tests are included at "test\test_cases\cp3tests".

Extra features and Known Bugs

Compiler Options We added a few extra options to our compiler. Full details can be found in the man page:

- `-z` - Do not invoke the Asc interpreter.
- `-p` - Print to stdout, like most other Unix command line applications.
- `-e` - Enable the language extensions. This enables ascDump() and ascTrace(), which are useful for debugging code. They map to the Asc debugging intructions; the default trace amount is 20 instructions.

Known bugs Our compiler is relatively free of bugs, however, here is a list of the bugs we know about:

- `Nesting depth` - Our compiler can generate code upto a maximum lexical level of 16, or the number of display registers available. After this, undefined behaviour will occur. Note that this is NOT recursive depth; this is lexical depth. However, while this is not an ideal situation, only in VERY rare cases would it be useful to have nested functions beyond a lexical depth of 16.
- `Math builtin Inaccuracy` - Our library functions are not always accurate in their return value. This is especially true for the sin function; inputs larger than 10 become very, very inaccurate. This is most likely a limitation in the Asc interpreter, it was observed to be off by over 50 on large floating point numbers. As well, most of the library functions were tested in Python, and output much more accurate results than their Asc counter-parts.
- `Builtin function hang` - Some inputs will cause some of the library functions to hang indefinitely. This is the case for ln(4) and some others. This may be related to inaccuracy of floating point numbers with the interpreter, however, it may also be implemenation flaws of our library functions.