PAL Compiler Documentation

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

November 8, 2013

Overview

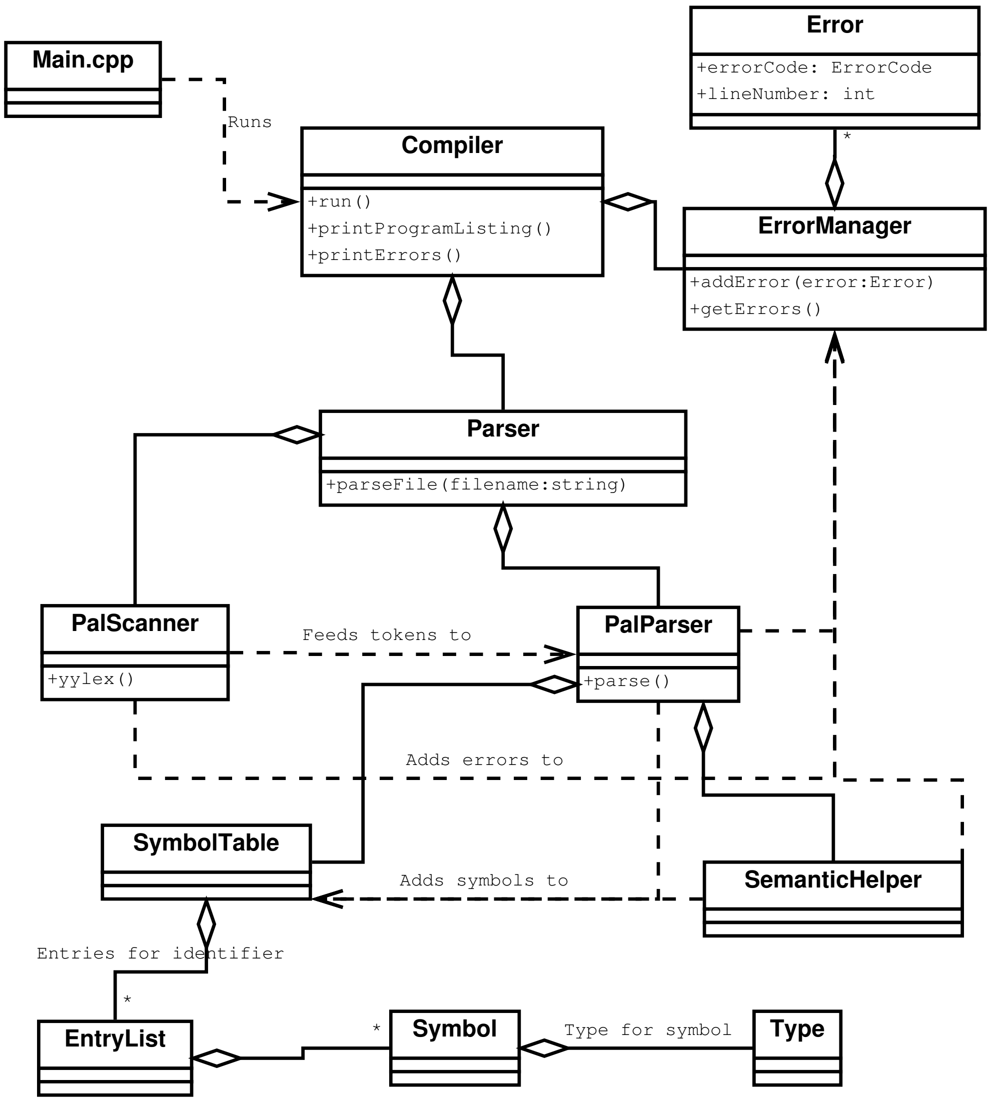

Figure 1: General structure of the compiler's parsing and semantic analysis components.

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...

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)

- `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). This made it very easy to add test cases and to run them while making changes to the grammar or token identifiers.

Submitted Tests 0.pal - 9.pal Tests 0-1 contain 3 errors which are all syntactical. Tests 2-7 contain only semantic errors. Test 8 is essentially the output of a dictionary. The program header is correct. Test 9 checks tricky nested declarations. These tests are included at "test\test_cases\cp2tests".