Pascal to ARM Compiler Report

Contents

Introduction

The compiler's core functionality is to convert basic Pascal Code into ARM assembly file. There is support for the most basic aspects of Pascal.

Extent of Language Support

The compiler generally conforms to the Pascal language described in the IEC/ISO 7185 standards (*referred to onwards as ISO Pascal)*. It also supports some minor additions by other Pascal extensions. *Free Pascal Reference* (Canneyt, 2011) was used as the reference for the syntax and contextual handling of Pascal.

Table 1 will summarise the list of features the compiler supports:

Table 1 Supported language features.

Feature Description	Remarks
Comments	Supports the delimiters: • { … } • {* … *} • /* … */ • //
Variable Types	
Integers	
Subrange	Subrange is treated simply as though it is of its base type. There is no boundary checking.
Character	Characters are simply stored as integers.
Pointers	Pointer arithmetic (which was not in the ISO Pascal) is not supported. NOTE: Pointers to arrays are possible to construct in the program but are untested
Arrays	NOTE: Arrays of pointers are possible to construct in the program but are untested
Functions and Procedures	
Functions/Procedures	Supported. Nested and recursive calls should be possible but under certain circumstances, the memory manager might get confused about the parameters and return values of the functions/procedures.
Parameters	Up to three parameters along with one return value is supported. The compiler will not complain if more than three parameters are used but only the first three parameters will be processed. Support for VAR (pass by reference) parameters is available. There are cases where the memory manager might get confused about the parameters and the return registers.
Expressions	
Full Expressions Support	The parser and contextual handler should be able to

	handle arbitrarily complex expressions with caveats: • The parsing and execution of complex expressions will require the use of temporary variables. • If these temporary variables get too numerous (more than the number of available registers), the memory manager will discard them and results might be lost. The parser supports the usual arithmetic operators and some relational operators. The parser also has limited optimisation, as will be described later.
Statements	
Assignment Statements	Assignment to normal variables, pointers dereferenced and array entries are supported.
Procedure Statement	Supported, though nested procedure calls should be possible, but are untested.
If Else Statements	If and Else are tested. Nested If and else statements should be possible, but are untested.
For Statements	Supported. Nested for loops or other statements are supported but not tested.
While Statements	Supported. Nested statements are possible but not tested.

The parser will parse and accept syntactically some of the unsupported feature of the language (like records and enums) but no contextual handling of these codes will take place. The parser will simply parse them and discard their input. In some cases, the compiler will run into a segmentation fault when an unsupported feature is used but is accepted by the parser. The use of *Labels* and *Goto* will cause the parser to spew out a warning message.

Program Overview

There are three main components to the compiler: the lexer, the parser and the program context manager.

Tools Used

The lexer was generated using the tool *Flex* while the parser was generated using a tool called *Bison*. *Make* is used to compile the compiler using *GCC*. Finally, the compiler makes extensive use of a new feature of the C++11 standard template library: shared_ptr[1]. This allows for the liberal use of pointers[2] to pass objects around without worrying about memory leaks. These pointers will automatically perform garbage collection when no more references to the pointers exist.

[1] See http://en.cppreference.com/w/cpp/memory/shared_ptr for a description of its API.
[2] Pointers in this literature will refer to the new smart pointers provided by C++11, specifically shared_ptr.

Compiler Execution Flow

There are three stages in the execution of the compiler: lexical analysis, parsing and code generation. Lexical analysis is done by the lexer which will walk through the source file, identifying *Tokens* using regular expression patterns. These *tokens* are then passed to the *parser* which will determine if the tokens correspond to a set of grammar rules known as production rules. The actions in these production rules will generate intermediate ARM Assembly code and establish the context of the program in the form of symbols using the context manager. Finally, when the parser is done with the file, it will call the context manager to generate the final ARM code output. This execution flow is shown in Figure 1.

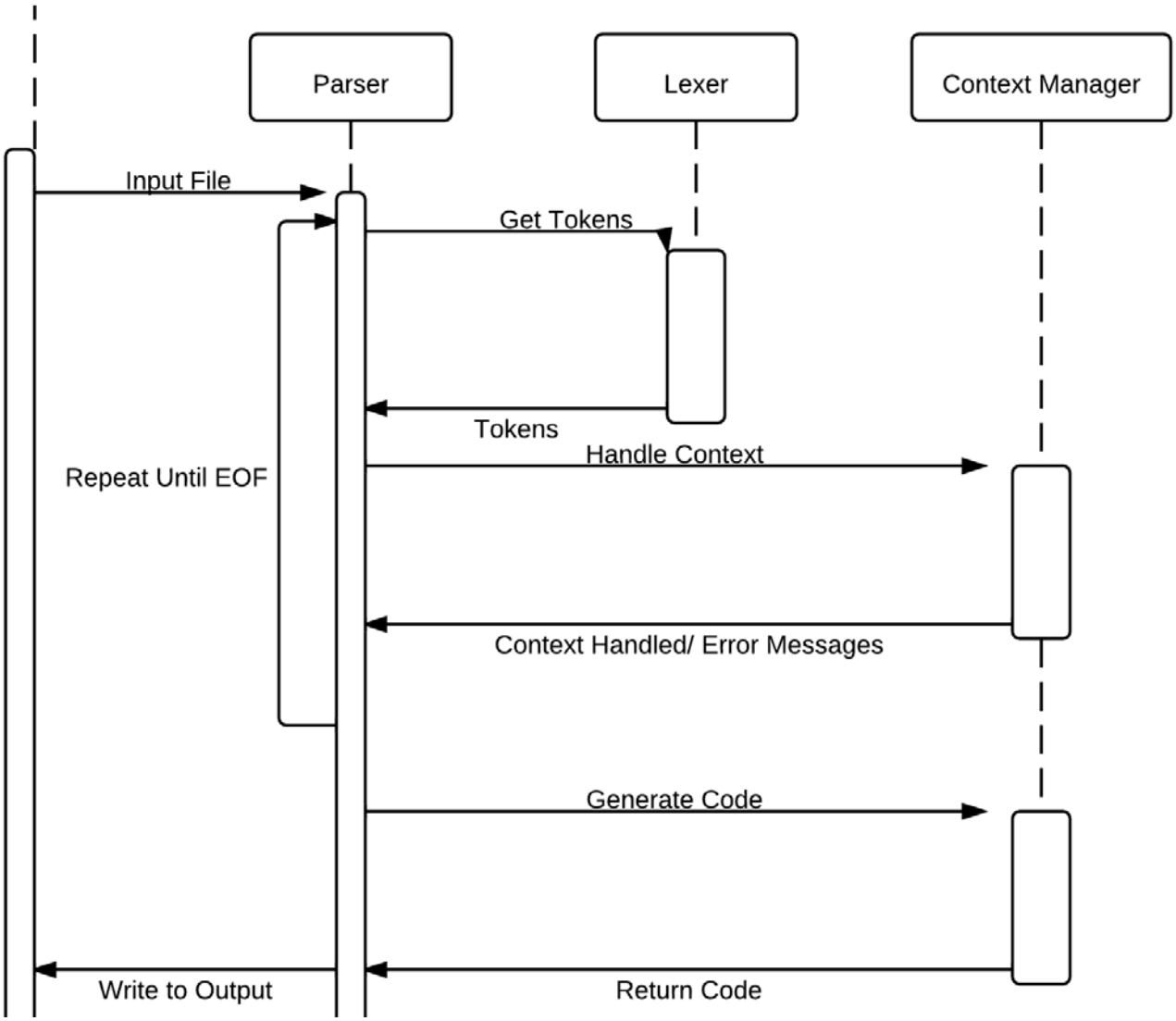


Figure 1 - Program execution flow.

Lexical Analysis Phase

Using regular expressions pattern defined in the file "*lexer.l*", Flex will generate a parser that will identify tokens in the source file based on the regular expression patterns. The actions in each case will create a pointer containing an object either of Token class or any of Token's derived classes. The lexer will then assign this pointer to the global variable "*yylval*" which will be read by the parser. It will then return the type of token found to the parser.

In an auxiliary function, the lexer also stores and advances the line and character number of the character it is processing. This is passed on to the parser via the tokens for use in error message generation. The lexer is also in charge of stripping away comments. Any "illegal" characters will be detected by the lexer and an error message will be generated.

Token Class

The token class will encapsulate the necessary attributes and data of the tokens found to be passed to the parser. All pointers to objects assigned to "*yylval*" or passed among production rules in the parser will be a pointer to an object of type Token. Because C++ allows for pointers of base class to point to a derived class, dynamic or static casts[3] will be used to convert the pointers to their desired types.

A summary of the classes and their relations can be seen in the UML diagram in *Annex A*. A summary of the description of each class can be found in Table 2.

The Token class itself serve as the base class for all the other Token classes. This class provide the basic functions and data members to store string-based tokens such as identifiers. It also defines some commonly used functions as virtual so that there is less need to cast the pointers should any of the derived class need to override functionality. However, due to the differing needs of each derived class along with the different function parameters and return type, polymorphism is not greatly exploited and instead, pointers are frequently casted.

Class Descriptions

The lexer will only generate objects of types *Token*, *Token_Int* or *Token_Char*. The rest of the Token classes are generated by the parser based on the syntactical arrangement of the token types and tokens found by the lexer. They are all summarised in Table 2.

Table 2 Description of the Token classes.

Name	Description
Token	Base Class
Token_Int	Store integer and Boolean literals.
Token_Char	Store character literals.
Used and Generated by Parser	

[3] Because of using shared_ptr, the new template functions dynamic_pointer_cast or static_pointer_cast are used instead.

Token_Func	Token representing a function along with its parameters.
Token_FormalParam	The formal parameters of a function.
Token_Expression Token_SimExpression Token_Term Token_Factor	Represent the parts of an expression
Token_ExprList	A wrapper for a vector of expressions.
Token_Var	A variable.
Token_Type	A type.
Token_IDList	A wrapper for a vector of identifiers stored in Token objects.

Parsing Phase

The parser function *yyparse* is invoked by the *main* function to begin compilation of the source code. It repeatedly requests tokens from the lexer. The parser is generated by the tool *Bison*. The parser exists as a finite state machine (FSM) where, depending on the input token type, the FSM will proceed to the respective state. If an input is invalid for a particular state, the parser will generate an error message accordingly.

The parser is not aware of the context of the program. While a statement may be syntactically correct, the context might be wrong. For example, adding an integer to a Boolean is correct syntactically but the parser is not aware of the type of the expressions involved and will not detect this as an error.

The FSM is generated by Bison using the *production rules* specified in the "*parser.y*" file. Each production rule has an action associated. This action usually calls the context manager to do context checking and to generate intermediate code[4]. When the parser reaches the end of file, it will call the context manager to generate the final ARM assembly code.

Symbol Generation and Access

When the parser encounters a variable, procedure, function or type declaration, it will proceed to ask the context manager to generate the appropriate symbols. The context manager will check for duplication and legality of the operation and then proceed to generate the necessary intermediate code.

When the parser encounters a reference to a variable, procedure, function or type, it will ask the context manager to check its internal symbol table for the legality of the access and then process it accordingly.

[4] As will be explained in a later section, the context manager does not really generate intermediate code per se.

Expressions

Due to the recursive nature of expressions and the possibility of optimisation of constant expressions (such as *3+5*), the parser will build a tree of the expression using the context manager. The specifics of the tree will be described in a later section.

When the tree is complete, the parser will call on the context manager to generate intermediate code for the expression in a process known as *flattening*. This process will also be described in a later section.

Scope Change

In Pascal and based on what features of Pascal that has been implemented, a scope change will only take place in the subroutine block of a function or procedure. Neither the parser nor the context manager takes note of this context change during the parsing phase. The parser will generate an internal operation via the intermediate code to push or pop the *AsmBlock* object containing the scope in a stack.

Intermediate Code

The context manager does not generate intermediate code per se – in fact it generates ARM assembly code encapsulated in *AsmLine* objects with the operators encapsulated in *AsmOp* objects. The *AsmOp* objects will encapsulate the literals or any symbols that are referenced by the instruction. For example, a representation of the statement

```
A := 5 + B;
```

might be

```
ADD {A}, 5, {B}
```

where *{A}* and *{B}* are the symbols for their respective variables.

Internal operations such as memory allocations are represented in this "intermediate code" in the form of custom Operation Codes (OpCodes) such as "BLOCKPUSH".

Code Generation Phase

The *AsmRegister* class is a class that manages the state of the registers during the code generation phase. It will track the registers and the symbols the registers represent. The context manager will use the intermediate code and replace the internal representation of the operators via their register or immediate literal equivalent with the aid of the AsmRegister objects assigned to each scope.

During code generation, internal operations are also performed, though sometimes these internal operations only change the state of the AsmRegister object and do not actually generate any code.

Components

This section will go through some of the more complicated components of the compiler in more detail.

Expressions

The production rules of expressions are summarised below:

```
Expression ➔ Expression SimpleOp SimExpression | SimpExpression
SimpleOp ➔ < | > | <= | >= | = | <>
SimExpression ➔ SimExpression TermOp Term | Term
TermOp ➔ + | - | OR | XOR
Term ➔ Term FactorOp Factor | Factor
FactorOp ➔ * | / | DIV | MOD | AND
Factor ➔ VariableReference | ( Expression ) | Constant | NOT Factor
```

The entities Expression, SimExpression, Term and Factor each have their own representation in terms of tokens classes are listed in Table 2. The tokens will form a tree according to the production rules above with *Expression* as the root of the tree.

During the flattening of the tree, temporary variables are used to calculate some of the operations. If any of these operations can be evaluated to a constant expression at compile time, they will be calculated. For example, the expression 5+3 will generate *MOV R0, #8* instead of the ADD operation.

Context Manager

There are seven classes that make up the context manager and a summary of their relations is shown in Annex B.

Symbols

Symbols represent the types, functions, procedures or variables of the program. All the symbols must have unique names in their respective scope, with the scope represented by AsmBlock objects. The symbols are held in AsmBlock objects in a hash map.

Symbols may be duplicated during code generation to change certain properties such as dereferencing of pointers. This does not affect the original symbol and is used to differentiate a pointer from a dereferenced pointer for the purpose of code generation.

AsmFile

AsmFile is the "glue" class that binds all the rest of the context manager class. The parser will use an instance of AsmFile to call almost all the operations during the parsing phase to generate intermediate code. It holds lists of the intermediate code (in the form of AsmLine objects) and all the scopes (in the form of AsmBlock objects). It also contains the algorithm for code generation and flattening of expression trees.

AsmBlock

Each AsmBlock is logically equivalent to a scope in the program. There is a global scope defined for the program, held in AsmFile. When a function or procedure is declared, a new AsmBlock object will be created via AsmFile. This object will be stored in AsmFile. Each AsmBlock will hold a list of Symbols of which all the names of the Symbols in the scope must be unique and not be the same as any reserved symbols defined in the global scope.

AsmBlock will also hold an instance of AsmRegister, the memory manager used during code generation by AsmFile.

AsmLine, AsmLabel and AsmOp

The intermediate code lines generated during the parsing face is held in the form of AsmLine objects. Each AsmLine object will hold the label for that line in the form of an AsmLabel object. Finally, the symbols or immediates are represented by the operators in the code line which are represented by AsmOp objects. These lines are stored and held by an AsmFile object.

AsmRegister

AsmRegister holds the state of the memory register during code generation phase. Each AsmBlock object will have an AsmRegister. AsmRegister will behave slightly differently if it belongs to a global scope or otherwise.

During code generation, AsmFile will request for registers to use via the AsmOp objects stored in the intermediate code (which are AsmLine objects). AsmRegister will then determine which register to use based on the state of the register at that point in time.

Other Components

There are some minor utility components in the compiler. Utility template functions to convert variables to strings and vice-versa are included. Due to the case insensitive nature of Pascal, all identifiers are stored internally in lowercase, therefore necessitating a utility function to convert characters to lowercase. Finally, there is a program argument parser that will do rudimentary argument parsing.

Running and Compiling the Compiler

Running

The compiler accepts arguments in the following syntax:

```
Compiler [flags] file
```

where *file* is the source file. If *file* is not provided, the compiler will use standard input as the input source.

The possible flags are:

`-o=file`

where *file* is the output file to generate ARM code to. If an output file is not specified, it will default to standard output.

`-s`

will force the compiler to generate code to save all registers to their respective memory locations at the end of each scope.

-p
will force the compiler to generate pedantic warnings during compilation.

Compiling

During testing, compilation was done under the following environment:

- Ubuntu 11.10
- GNU Make 3.81
- GCC 4.6.1
- Flex 2.5.35
- Bison 2.4.1

At least version 4.6.0 of GCC is required (for the use of nullptr).

Compilation can be done by invoking the "make" command when the working directory is set to the program directory. "make clean" will clean up the generated files. Finally, "make again" will force a full recompilation.

NOTE: The directory structure should be maintained when running. This is because the compiler expects the file in the "Asm" directly to be named exactly as they are. These files are the skeletal files used to generate the final code.

Emulating

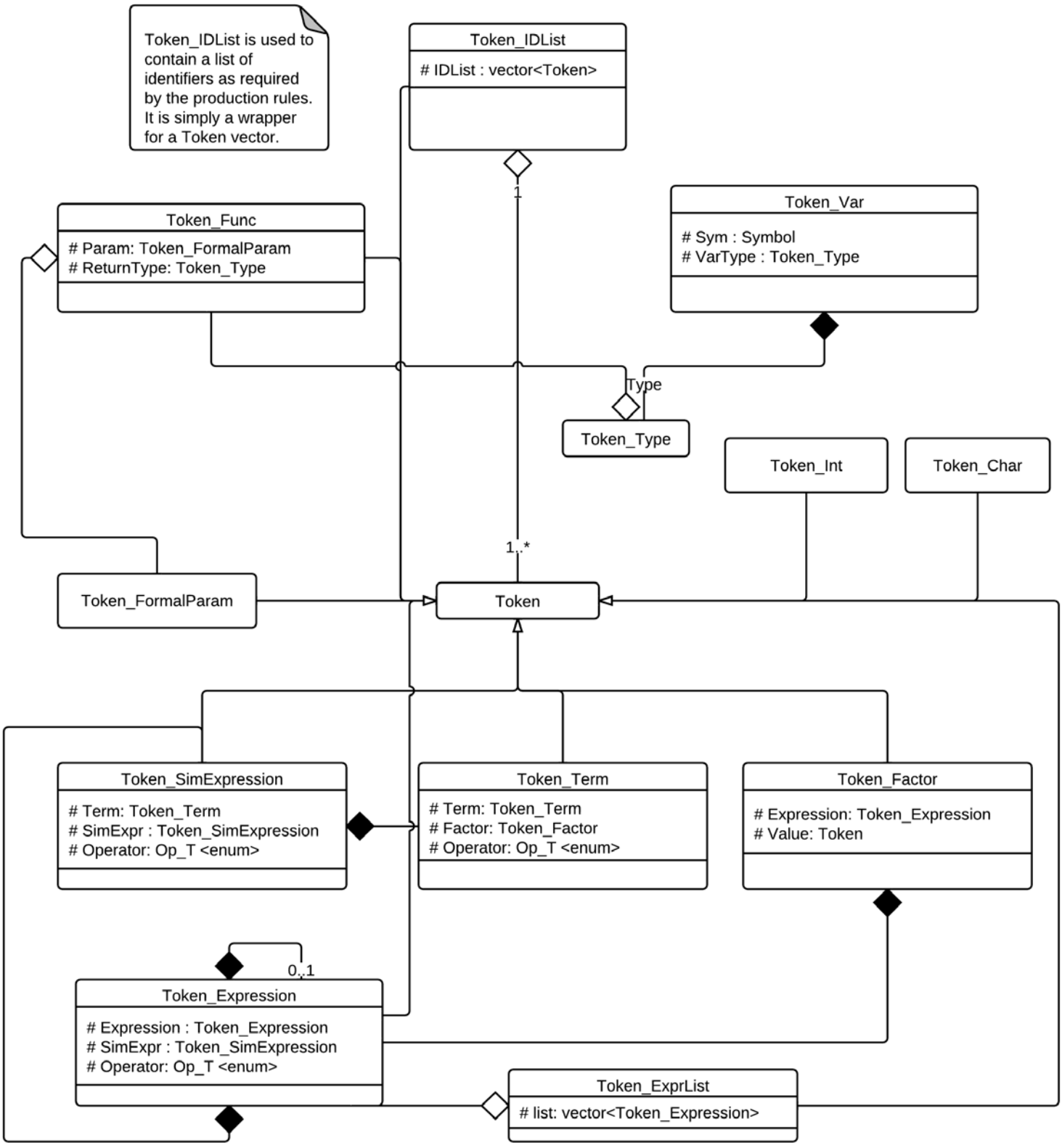
The Keil project file under the "Output" directory should give proper settings to compile the assembly file. The assembly will not emulate well in Keil due to the use of unsupported (by Keil) SWI calls.

Bibliography

Aho, A. V., Lam, M. S., Sethi, R. & D, J., 2006. *Compilers: Principles, Techniques, and Tools.* 2nd ed. s.l.:s.n.

Canneyt, M. V., 2011. *Free Pascal: Reference Guide.* [Online]
Available at: ftp://ftp.freepascal.org/pub/fpc/docs-pdf/ref.pdf
[Accessed 14 March 2012].

Annex A – Token Class Diagram



Annex B – Program Context Class Diagram