Introduction to language theory and compiling Project – Part 3

Antoine Passemiers – Alexis Reynouard

December 29, 2017

In this work, we implemented an actual compiler for the Imp language. Given the unambiguous LL(1) grammar (see previous report on how to produce it and how our program may help you to produce it) for the imp language and a source file, the compiler produces an LLVM-assembly language (LLVM IR) `.ll` file which may be compiled to LLVM bitcode using the LLVM assembler `llvm-as(1)`. So we extended the previously given Imp grammar to handle some new features and made improvements to the lexer and parser.

We will start by describing some choices we made to implement the compiler. Second we will introduce the new additional features of the language. Finally, we will present the new enriched Imp grammar, as well as the actual code generator and the additions to the lexer and parser.

Contents

1 Assumptions, limitations and implementation choices

First it worth noting that there exists two scopes in Imp. The global scope outside any function and the local scope inside the functions.

The Imp variables are translated into named LLVM IR variables. The temporary variables needed are implemented as unnamed LLVM IR variables.

The Imp language only supports the `int32` type. Any expression contains `int32` values and is evaluated as an `int32` value. Notably, any function which does not explicitly return a value will actually return 0. Indeed, since the original Imp grammar (as given in the project statements) only handles `int32` values, this does not induce any loss of generality. Also, allowing the use of void would imply that the result of void functions cannot be assigned to Imp functions, which require to make additional changes to our final LL(1) grammar.

If the code generator detects an inconsistency due to the usage of another grammar, an error message will be displayed and the program will exit with a -1 error code.

Note also that tests may be automatically run with the script `more/tests.sh`. The makefile is quite complete and provides a lot of way to run specific operations and run tests as well.

2 The enriched Imp language (bonus features)

Here we present briefly the additions made to the Imp language. For a more precise description the reader can go to the next section.

2.1 Randomness

The ability to use pseudo random numbers was added to the language through the `rand` keyword. The expression `rand(a)` assigns a pseudo random value between -2,147,483,647 and 2,147,483,647 to the variable `a`.

2.2 Functions

Informally, an Imp function is an executable statement, composed of any instruction list without `define` and `import`, and whose the value is an `int32`.

A function must be defined before any usage. The way to define a function is described by the variable `<Define>` of the language. A usage of, or "call to", the function is defined by the `<Call>` variable. The arguments are passed to the function by copy and the inner variable are defined in a local scope. Also note that user-defined functions must start by a '@'. This has been inspired by the llvm syntax itself (for no valid reason) and facilitated the maintenance of our symbol table and lexer implementation.

To take an example, the following program print 12:
`begin function @foo(arg) do print(arg) end; @foo(12) end`

Because a function has to be defined before its first call, recursive calls are not supported.

Unfortunately for now the compiler is not able to detect that a function call respect the function signature, this can lead to an error when running the llvm assembler `llvm-as`, like

```
llvm-as: main.ll:85:19: error: '@foo' defined with type 'i32 (i32*, i32*)*'
    %2 = call i32 @foo(i32* %c0c0)
```

Here for example, the function was defined with two parameters but called with only one argument.

2.3 Import

The `import` keyword has been added to the language, and offers the possibility to include pre-compiled llvm code to the output llvm program. A file designed to be `import`ed into other files is called a module and must be written in llvm. Also, a module can obviously not contain any main function because it is designed to be included in a file that already provides one.

2.4 The standard library management

The standard library provides three modules. Two of them, `_stdio` and `_random` are imported by default. They provide some functions whose names are self-explanatory: `rand`, `getNumber`, `getchar`, etc. We implemented a `STDLibManager` that checks whether a module has already been loaded or not, and includes the pre-compiled code in the target file if that is not the case. To be able to keep track of which functions are included using the `import` keyword, we use minimalistic header files that only contain the function names (the stdlib manager does not check the function signature yet, but this can be easily done in future versions). Those header files have the *.sih* extension (which stands for Super Imp Header).

The third module is called `_math`, and contains the definitions of `min` and `max` functions. This is somewhat sparse but only intends to prove that it is easy to create new modules. We basically wrote the `_math` Imp module, compiled it to llvm, and included the file to the *imp_stdlib* folder. It would be quite easy to add a lot of new functions and modules.

3 Augmenting Imp's syntax and grammar

3.1 Regular expressions

The lexer was extended to read the new keywords: `rand`, `function`, `return`, `import`, `,` (comma), as well as two new "kinds of identifier": FuncName and ModuleName which are respectively identifiers for functions (begins with a "@") and identifiers for modules (begins with a "_").

```
Identifier     = {Alpha}{AlphaNumeric}*
FuncName       = @{Identifier}
ModuleName = _{Identifier}
```

Please also note the trivial regular expressions for detecting the new Imp keywords: *rand, function, return, import* but also ",".

3.2 Grammar rules

Here is the new grammar for the Imp language. Main additions and changes are designated by colored rules and variables.

One can note four main changes:

- <Instruction> has been split into <Instruction> and <FuncInstruction> (and some other variables have been updated to reflect this change). <FuncInstruction> is defined as an <Instruction> without <Define> and <Import>.
- <ParamList> has been introduced to define functions (with <Define>),
- <ArgList> has been introduced to call functions (with <Call>),
- <Assign> has been modified to allow assigning a variable with the result of a function call.

[1]	<Program>	→ begin <Code> end
[2]	<Code>	→ <InstList>
[3]	<InstList>	→ epsilon
[4]	<FuncInstList>	→ epsilon
[5]	<Instruction>	→ <Define>
[6]		→ <Import>
[7]		→ <FuncInstruction>
[8]	<FuncInstruction>	→ <Assign>
[9]		→ <If>
[10]		→ <While>
[11]		→ <For>
[12]		→ <Print>
[13]		→ <Read>
[14]		→ <Rand>
[15]		→ <Return>
[16]		→ <Call>
[17]	<Define>	→ function [FuncName] (<ParamList>) do <FuncInstList> end
[18]	<Return>	→ return <ExprArith-p0>
[19]	<Import>	→ import [ModuleName]
[20]	<Call>	→ [FuncName] (<ArgList>)
[21]	<ArgList>	→ epsilon
[22]	<ParamList>	→ epsilon
[23]	<ExprArith-p0-j>	→ <Op-p0> <ExprArith-p1>
[24]	<ExprArith-p0-i>	→ <ExprArith-p1>
[25]	<ExprArith-p1-j>	→ <Op-p1> <Atom>
[26]	<ExprArith-p1-i>	→ <Atom>
[27]	<Atom>	→ [VarName]
[28]		→ [Number]
[29]		→ (<ExprArith-p0>)
[30]		→ - <Atom>
[31]	<Op-p0>	→ +
[32]		→ -
[33]	<Op-p1>	→ *
[34]		→ /
[35]	<Cond-p0-j>	→ or <Cond-p1>
[36]	<Cond-p0-i>	→ <Cond-p1>
[37]	<Cond-p1-j>	→ and <Cond-p2>
[38]	<Cond-p1-i>	→ <Cond-p2>
[39]	<Cond-p2>	→ not <SimpleCond>
[40]		→ <SimpleCond>
[41]	<SimpleCond>	→ <ExprArith-p0> <Comp> <ExprArith-p0>

[42]	<Comp>	→ =
[43]		→ >=
[44]		→ >
[45]		→ <=
[46]		→ <
[47]		→ <>
[48]	<While>	→ while <Cond-p0> do <FuncInstList> done
[49]	<Print>	→ print ([VarName])
[50]	<Read>	→ read ([VarName])
[51]	<Rand>	→ rand ([VarName])
[52]	<InstList>	→ <Instruction> <InstList-Tail>
[53]	<InstList-Tail>	→ ; <InstList>
[54]		→ epsilon
[55]	<FuncInstList>	→ <FuncInstruction> <FuncInstList-Tail>
[56]	<FuncInstList-Tail>	→ ; <FuncInstList>
[57]		→ epsilon
[58]	<ArgList>	→ <ExprArith-p0> <ArgList-Tail>
[59]	<ArgList-Tail>	→ epsilon
[60]		→ , <ArgList>
[61]	<ParamList>	→ [VarName] <ParamList-Tail>
[62]	<ParamList-Tail>	→ epsilon
[63]		→ , <ParamList>
[64]	<Assign>	→ [VarName] := <Assign-Tail>
[65]	<Assign-Tail>	→ <Call>
[66]		→ <ExprArith-p0>
[67]	<If>	→ if <Cond-p0> then <FuncInstList> <If-Tail>
[68]	<If-Tail>	→ endif
[69]		→ else <FuncInstList> endif
[70]	<For>	→ for [VarName] from <ExprArith-p0> <For-Tail>
[71]	<For-Tail>	→ to <ExprArith-p0> do <FuncInstList> done
[72]		→ by <ExprArith-p0> to <ExprArith-p0> do <FuncInstList> done
[73]	<ExprArith-p0>	→ <ExprArith-p0-i> <ExprArith-p0-j>
[74]	<ExprArith-p0-j>	→ epsilon
[75]	<ExprArith-p1>	→ <ExprArith-p1-i> <ExprArith-p1-j>
[76]	<ExprArith-p1-j>	→ epsilon
[77]	<Cond-p0>	→ <Cond-p0-i> <Cond-p0-j>
[78]	<Cond-p0-j>	→ epsilon
[79]	<Cond-p1>	→ <Cond-p1-i> <Cond-p1-j>
[80]	<Cond-p1-j>	→ epsilon

4 Implementation

4.1 Improvements in both lexer and parser

To read the new keywords, the lexer was enriched with the corresponding lexical units: `RAND`, `FUNCTION`, `FUNCNAME`, `RETURN`, `IMPORT`, `MODULENAME` and `COMMA`.

The parser has undergone improvements as well as some modifications, especially to display errors in a more convenient way. The error management has been improved and described in a following section.

4.2 LLVM code generator

We designed both lexer and parser in such a way that it minimizes the maintenance cost when adding new Imp features. When new grammar variables and rules (at the condition that the new grammar is unambiguous), the grammar is automatically made LL(1), and the action table is automatically updated. For this part of the project, we decided to implement our code generator in a recursive descent fashion: this allowed us to implement the code generation directly inside mutually recursive methods. As a result, the only part of the Imp compiler that is not modular (due to hard-coding) is located at the same place where maintenance is mandatory for generating actual llvm code.

For each variable of the Imp grammar, there is a corresponding hard-coded method in *CodeGenerator* that generates the appropriate llvm code. When needed, such a method returns an unnamed variable (which is a string representation of an integer) used to store the result of the code generated. For example, when working on `a := 7 + 9`, 7 and 9 are both stored in an unnamed llvm variable. The latter is defined and returned by the function which generates the corresponding code (`generateFromAtom` in this case). Then, the result of the addition is stored in an unnamed variable too (`generateFromExprArithPOJ`). The reasoning holds for all grammar variables with intermediary results.

4.3 Error handling

The compiler may report some errors due to an input which does not correspond to the given grammar, or which may not produce some valid code (call to an undefined function for example). Also, the compiler is able to detect errors in the grammar given as input (see the previous report).

Each type of error in the source file is detected by one of the compiler component, according to the kind of error.

The Lexer may report an error (`BadTerminalException`) when a terminal can not be associated to a lexical unit, or sometimes when a terminal is encountered in a bad context (`BadTerminalContextException`) (see first report).

The parser report errors either when the `read` token (for example) does not yield any action in the action table according to the current top of stack (`UnexpectedSymbolException`), or when the end of file is encountered before the end of the program (`UnexpectedEndOfFileException`).

The code generator reports an error when one tries to call an undefined function (`UndefinedFunctionException`). Note that there is no declaration required in Imp for (`int32`) variables. So all the variables are implicitly declared at the beginning of the program, but not assigned. It is the responsibility of the programmer to ensure that a value is assigned to the variable before use, otherwise the behaviour of the program may be unexpected.

Returned error codes On success, the compiler returns 0 (or the success value of the platform). Otherwise, the return value depends on the error. These values are shown in the table below:

Error from	code	description
OS	1	IO error
	2	File not found
Lexer	16	Unkown terminal
	17	Bad terminal context
Parser	19	Unexpected symbol
		Unexpected end of file
Code generator	18	Call to undefined function
Main class	-1	Unknown
	-1	Not imp grammar[1]

Here is an example of error reported by the parser when one try to define a function inside another function (nested functions are not supported by our Imp compiler):

[1]Temporary error because of the modular lexer and parser (work well with any ll(1) grammar whose keywords and lexical units are the same as the current Imp language) working with a currently non-modular code generator (the Imp grammar is hard-coded).

```
Unexpected symbol:
'function' with TOS: '<FuncInstList>'
LL1Parser:
    token number: 35
    stack: [ '<FuncInstList>', 'end', '<InstList-Tail>', 'end', '$' ]
    token: function     lexical unit: FUNCTION
    line: 5, column: 1
    code: + 1 ; function @bar ( b
```