Introduction to language theory and compiling
Project – Part 3

Antoine Passemiers – Alexis Reynouard

December 26, 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 the program may help) for the imp language and a source file, the compiler produces an LLVM-assambly language (LLVM IR) `.ll` file which may be compiled to LLVM bitcode with 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 begin with some choices we did to implement the compiler. Second we will introduces the new additional features of the language. Finally, we will present the new enriched Imp grammar, then the actual code generator and the additions to the implementation of 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 support the `int32` type. Any expressions which have a value have a `int32` value. Notably, any functions which does not explicitly return a value returns 0.

The usage may seems strange because we have to give to the compiler the grammar of the Imp language. This is because we implemented modular lexer and parser, but the code generation was hard-coded due to a lack of time. If the code generator detect an inconsistency due to the usage of another grammar, an error message will be displayed and the program exit with the error code -1.

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

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 TODO: min and TODO: max 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.

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

One can create a file to contains code for later use in other files. Once created any file may be imported in another thanks to the `import` keyword. TODO:

A file designed to be `imported` into other files is called a module.

2.4 The standard library

The standard library provide three modules. Two of them, `TODO:` and `TODO:` are imported by default. They provide some functions whose the name is self-explanatory: `TODO:`

The last one may be included with `TODO:` and provide the `TODO:` and `TODO:` functions.

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}
```

3.2 Grammar rules

Here is the new grammar for the Imp language. TODO: The main additions and changes are marked with an asterisk (*).

One can note four main changes:

- <Instruction> is divided into <Instruction> and <FuncInstruction> (and some other variables were updated to reflect this change). <FuncInstruction> is defined as an <Instruction> without <Define> and <Import>.
- <ParamList> was introduced to define functions (with <Define>),
- <ArgList> was introduced to call functions (with <Call>),
- <Assign> was modified to allow to assign 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 receive the corresponding improvement as well as some modifications, especially to display errors in a more convenient way.

The error management was improved as described in a following section.

4.2 LLVM code generator

For each variable of the grammar, a method to produce the corresponding llvm code is hard-coded. When needed, such a method return the number of an unnamed variable 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. Which one is defined and returned by the function which generate the corresponding code (`generateFromAtom`). Then, the result of the addition is stored in an unnamed variable too (`generateFromExprArithPOJ`).

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). Sometimes the compiler may detect an error in the grammar (see the previous report).

Errors in the source file are 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 the first report).

The parser report errors either when the read token does not give 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 report an error if one try 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 return 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
	-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:

```
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
```

[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 not modular code generator (the Imp grammar is hard-coded).