Introduction to language theory and compiling
Project – Part 3

Antoine Passemiers – Alexis Reynouard

December 24, 2017

In this work, we implemented an actual compiler for the Imp language.

Given an unambiguous LL(1) grammar (see previous report on how to produce it and how the program may help) 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 Imp does not know about scopes. All the variables are global. TODO: and for functions arguments?

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, all functions which does not explicitly return a value return 0.

2 The enriched Imp language (bonus features)

2.1 Functions

Imp functions are a way to group instructions together to later executions. They are executed

- Fonctions (definition et appel, arite au choix)
- Mot cles rand, import, function
- Gestion de librairie standard
- Gestion d'exceptions
- Optimization (faudrait qu'on checke les flags de llvm pour qu'il optimise les trucs immondes qu'on genere)
- Si tu trouves des trucs facile a ajouter, ne te prives pas

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. The main additions and changes are marked with an asterisk (*).

One can note tree main changes:

- <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]		→ epsilon
[4]	<Instruction>	→ <Assign>
[5]		→ <If>
[6]		→ <While>
[7]		→ <For>
[8]		→ <Print>
[9]		→ <Read>
*[10]		→ <Rand>
*[11]		→ <Define>
*[12]		→ <Return>
*[13]		→ <Call>
*[14]		→ <Import>
*[15]	<Define>	→ function [FuncName] (<ParamList>) do <Code> end
*[16]	<Return>	→ return <ExprArith-p0>
*[17]	<Import>	→ import [ModuleName]
*[18]	<Call>	→ [FuncName] (<ArgList>)
*[19]	<ArgList>	→ epsilon
*[20]	<ArgList>	→ <ExprArith-p0> <ArgList-Tail>
*[21]	<ArgList-Tail>	→ epsilon
*[22]		→ , <ArgList>
*[23]	<ParamList>	→ epsilon
*[24]	<ParamList>	→ [VarName] <ParamList-Tail>
*[25]	<ParamList-Tail>	→ epsilon
*[26]		→ , <ParamList>
*[27]	<Assign>	→ [VarName] := <Assign-Tail>
*[28]	<Assign-Tail>	→ <Call>
*[29]		→ <ExprArith-p0>
[30]	<Atom>	→ [VarName]
[31]		→ [Number]
[32]		→ (<ExprArith-p0>)
[33]		→ - <Atom>
[34]	<Op-p0>	→ +
[35]		→ -

[36]	<Op-p1>	→ *
[37]		→ /
[38]	<Cond-p0-j>	→ or <Cond-p1>
[39]	<Cond-p0-i>	→ <Cond-p1>
[40]	<Cond-p1-j>	→ and <Cond-p2>
[41]	<Cond-p1-i>	→ <Cond-p2>
[42]	<Cond-p2>	→ not <SimpleCond>
[43]		→ <SimpleCond>
[44]	<SimpleCond>	→ <ExprArith-p0> <Comp> <ExprArith-p0>
[45]	<Comp>	→ =
[46]		→ >=
[47]		→ >
[48]		→ <=
[49]		→ <
[50]		→ <>
[51]	<While>	→ while <Cond-p0> do <Code> done
[52]	<Print>	→ print ([VarName])
[53]	<Read>	→ read ([VarName])
*[54]	<Rand>	→ rand ([VarName])
[55]	<InstList>	→ <Instruction> <InstList-Tail>
[56]	<InstList-Tail>	→ ; <InstList>
[57]		→ epsilon
[58]	<If>	→ if <Cond-p0> then <Code> <If-Tail>
[59]	<If-Tail>	→ endif
[60]		→ else <Code> endif
[61]	<For>	→ for [VarName] from <ExprArith-p0> <For-Tail>
[62]	<For-Tail>	→ to <ExprArith-p0> do <Code> done
[63]		→ by <ExprArith-p0> to <ExprArith-p0> do <Code> done
[64]	<ExprArith-p0-j>	→ <Op-p0> <ExprArith-p1>
[65]	<ExprArith-p0-i>	→ <ExprArith-p1>
[66]	<ExprArith-p1-j>	→ <Op-p1> <Atom>
[67]	<ExprArith-p1-i>	→ <Atom>
[68]	<ExprArith-p0>	→ <ExprArith-p0-i> <ExprArith-p0-j>
[69]	<ExprArith-p0-j>	→ epsilon
[70]	<ExprArith-p1>	→ <ExprArith-p1-i> <ExprArith-p1-j>
[71]	<ExprArith-p1-j>	→ epsilon
[72]	<Cond-p0>	→ <Cond-p0-i> <Cond-p0-j>
[73]	<Cond-p0-j>	→ epsilon
[74]	<Cond-p1>	→ <Cond-p1-i> <Cond-p1-j>
[75]	<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`.

4.1.1 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.

TODO: returned error codes

TODO: example

```
Unexpected symbol:
'[VarName]' with TOS: '[FuncName]'
LL1Parser:
    token number: 2
    stack: [ '[FuncName]', '(', '<ParamList>', ')', 'do', '<Code>', 'end', '<InstList-Tail>', '
    token: foo  lexical unit: VARNAME
    line: 2, column: 9
    code: begin function foo ( ) do print
```

4.2 LLVM code generator

recursive descent code generator ?

On a hard code une methode par variable de la grammaire. Quand c'est necessaire, une methode renvoie le nom d'une variable temporaire (non nommee) qui stocke le resultat de l'expression.

Exemple: a := (7 + 9)

7 est stockée dans une variable non nommee, puis de meme pour 9, puis 7 + 9 encore dans une autre, puis cette derniere est stockee dans la variable %a avec un store.