Introduction to language theory and compiling
Project – Part 3

Antoine Passemiers – Alexis Reynouard

December 23, 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.

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 Bonus features

- 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

arglist, paramlist... assign -> call

4 Implementation

4.1 Improvements in both lexer and parser

4.1.1 Error handling

erreur de syntax, nom de module non existant, fonction non definie...

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.