CC 0815

JAVA Compiler

Compiler Construction SS 2013

Alexander Gruschina - Mario Kotoy
Department of Computer Sciences
University of Salzburg
Salzburg, Austria

July 12, 2013

Contents

List of Tables

Listings

1 Project overview

This compiler is the result of a project we ran in a course called *Introduction to Compiler Construction*[1] at the Department of Computer Sciences at the University of Salzburg. Our instructor was Prof. Christoph Kirsch and our work is based on his lecture notes and videos.
CC 0815 is a *single pass Java* compiler, developed with *Eclipse*, hosted on *GitHub* and built on *drone.io*, further discussions during this course were held on *Piazza*. If you have any questions or requests you can contact us via email, addresses can be found at the end of this document.

2 Key features

- Global variables
- Local variables
- Delayed code generation
- Local hiding
- Procedures with or w/o return value
- Procedures with or w/o parameters
- Boolean expressions
- Lazy evaluation
- Nested loops and conditionals
- Arrays

Not fully implemented:

- Strings
- Self compilation
- Seperate compilation

- Records (which are in fact inner classes with no object-oriented behaviour)
- Encapsulated array and record structures
- Single line and multi line comments
- Error handling
- Type checking
- Evaluation of complex arithmetic and boolean expressions
- Output stream (System.out)

3 Scanner

The scanner is given a string, which represents the input file to be read from. When the method getToken() is called, the scanner reads the file character by character until he recognize a token or end of file. The comments are skipped and new lines are recognized to count the line number.

3.1 Token

A *Token* can be:

- a keyword:
  The keywords of our language are:
  - *int* ID = 0
  - *char* ID = 1
  - *string* ID = 2
  - *boolean* ID = 3
  - *void* ID = 4
  - *static* ID = 5
  - *class* ID = 6
  - *while* ID = 7
  - *if* ID = 8
  - *else* ID = 9
  - *true* ID = 10
  - *false* ID = 11
  - *new* ID = 12
  - *public* ID = 13
  - *return* ID = 14
  - *system* ID = 15
  - *null* ID = 16
- an operator: The operators of our language are:
  - ”(” ID = 50
  - ”)” ID = 51
  - ”{” ID = 52
  - ”}” ID = 53
  - ”[” ID = 54
  - ”]” ID = 55
  - ”;” ID = 56
  - ”.” ID = 57
  - ”,” ID = 58
  - ”:” ID = 59
  - ” ’” ID = 60
  - ” ”” ID = 61
  - ”+” ID = 62
  - ”-” ID = 63
  - ”*” ID = 64
  - ”/” ID = 65
  - ”>” ID = 66
  - ”<” ID = 67
  - ”!” ID = 68
  - ”&” ID = 69
  - ”|” ID = 70
  - ”=” ID = 71
  - ”>=” ID = 72
  - ”<=” ID = 73
  - ”!=” ID = 74
  - ”&&” ID = 75
  - ”||” ID = 76
  - ”==” ID = 77
- a number:
  An integer number ID = 101.
- an identifier:
  It’s a letter followed by letters or numbers and is no keyword ID = 102.
- a end of file token:
  This token indicates end of file ID = 99.

The symbol informations are stored in a global record variable token, which contains the fields:

- int symbolId:
  contains the ID's as given above.
- int number:
  if the symbol is a number the int value of the number is stored here.
- int lineNumber:
  the line number of the symbol for error handling.
- String identifier:
  if the symbol is a identifier the String representation of the identifier is stored here.

Listing 1: Token record

```java
class Token{
  int symbolId;
  int number;
  int lineNumber;
  String identifier;
}
```

4 Parser

The parser was rewritten two times, this enabled us a more detailed understanding of the components used in this *PDA*.Since we support native *Java* syntax, we designed our parser to fullfill these requirements.
Here are the formal properties of our parser:

- Recursive descent top-down parser
- *LL(1)*
- Parses our *context free* grammar as shown in section 4.1.
- If a parsing error occurs, symbols are skipped until syntactically correct code can be found
- Gives feedback in the form of *error* and *warning* messages

4.1 EBNF grammar

This is the *context free* grammar our *Parser* is able to parse.

Listing 2: CC 0815 grammar

```
Class        = "public" "class" Identifier "{" ClassBody "}" "EOF".
ClassBody    = {VarDec} {MethodDec}.
VarDec       = {(PrimDec | RecDec)}.
PrimDec      = Type ["[" "]"] Identifier ";".
Type         = "int" | "boolean" | "char" | RecType.
RecDec       = "class" Identifier "{" {PrimDec} "}".
MethodDec    = "public" ("void" | Type ["[" "]"]) Identifier
               "(" Param ")" "{" {PrimDec} {Statement} "}".
Statement    = "System" "." "out" "." "println" "(" Expression
               ["+" Expression] ")" ";"
              | Assignment ";"
              | Identifier "=" Expression ";"
              | ProcCall ";"
              | "while" "(" Expression ")" (Statement | StatemSeq)
              | "if" "(" Expression ")" (Statement | StatemSeq)
                ["else" (Statement | StatemSeq)]
              | "return" [Expression] ";".
Assignment   = Identifier [{("[" Expression "]" | "."
               Identifier)}] "=" (Allocation | Expression).
Allocation   = "new" Type ("(" ")" | "[" Expression "]").
StatemSeq    = "{" {Statement} "}".
Param        = Type ["[" "]"] Identifier [{"," Type
               ["[" "]"] Identifier}].
Expression   = ["-" Term] { ("+" | "-" | "==" | ">"
               | "<" | ">=" | "<=" | "||") Term.
Term         = Factor {("*" | "/" | "&&") Factor}.
Factor       = ProcCall
              | Identifier | Number
              | "(" Expression ")"
              | "!" Expression
              | "true"
              | "false".
ProcCall     = Identifier "." Identifier "(" [Expression
               [{"," Expression}]] ")".
Identifier   = Letter [{Letter | Digit}].
Number       = ["-"] Digit {Digit}.
Letter       = "a" | "b" | ... | "z" | "A" | "B" | ... | "Z".
Digit        = "0" | "1" | ... | "9".
```

5 Error handling

As discussed in the lecture, our compiler should be aware of two different types of errors:

- Symbol is missing
- Symbol is misplaced

In both cases, the user should get suited feedback. We handle this by using two different error output methods:

- *void errorAtToken (String expected, Token unexpected)*
- *void error (String message)*

As an example, if there is a missing right parenthesis as part of an expression, the parser knows at this point, it is an unexpected token and he calls *errorAtToken (String, Token)*. In this case the output looks as follows:

Listing 3: errorAtToken

```
+------------------------[ E R R O R ]------------------------
| Unexpected token received @ line 12
| Details:
|       token.identifier = {
|       token.symbolId = 52
|       token.number = 65535
| Expected: )
+-------------------------------------------------------------
```

Let's suppose the main class has a different name than the source file itself. In this case the user gets a more generalized message by calling error(String,Token):

Listing 4: error

```
+------------------------[ E R R O R ]------------------------
| class name should be same as file name.
+-------------------------------------------------------------
```

5.1 Warnings

As an additional feedback our compiler provides a method, which advises the user against different problems, that are not critical relating to compilation, though indicate an unwanted mistake. E.g., the user defines the following method:

Listing 5: Implicit type cast

```
public int getX (char x){
  return x;
}
```

The return type of this function is *int*, however, variable x is of type *char*. Actually, there is no problem because an implicit type cast from *char* to *int* occurs. Anyway, the user should get informed about this process, by calling *warning (String message)*:

Listing 6: warning

```
+----------------[ W A R N I N G ]----------------------
| Return type mismatch @ line 2
+-------------------------------------------------------
```

6 Target machine

Our CC 0815 interpreter reads code from a binary file, which was produced during parsing process. The target machine works with 32 32 bit registers and a memory of size 60 kB (byte aligned). As shown in the lecture, the memory is divided into Code, Global, Heap, Stack.

6.1 Code format

Basically we use the code format as shown in the lecture.

- Format *F1*: immediate addressing (e.g. ADDI, STW)
- Format *F2*: register addressing (e.g. ADD, MUL)
- Format *F3*: absolute jumps (e.g. JSR)

The parser generates code in binary format, stores it in a file, which is read by the target machine. Let's look at a very simple demo program:

Listing 7: A simple demo

```
public class Demo {
  public static void main(String[] args) {
    int i;
    i = 1;
    while (i < 10)
      i = i + 1;
  }
}
```

Parsing this demo will generate code as shown in table 1. The first instruction is always a *BR* to the main procedure. This address is retrieved at the end of compiling, by requesting the main procedure symbol node from the symbol table. Terminating the program is achieved by branching to the end of the file.

Table 1: executable code

HEX format	instruction	HEX format	instruction
44 00 00 01	BR 1	30 20 00 06	BGE 6
2B FE 20 00	PSH	04 20 00 01	ADDI
2B BE 20 00	PSH	1C 5D 00 00	LDW
53 A0 F0 00	ADD	50 42 08 00	ADD
0B DE 00 04	SUBI	20 5D 00 00	STW
04 20 00 01	ADDI	44 00 FF F8	BR -8
20 3D 00 00	STW	53 C0 E8 00	ADD
1C 3D 00 00	LDW	27 BE 20 00	POP
04 40 00 02	ADDI	27 FE 40 00	POP
64 21 10 00	CMP	44 00 00 01	BR 1

6.2 Instructions

Our instructions are based on the *DLX* instruction set, however we've extended this set to fit in with our compiler semantics. The target machine works with bytes, this means the offset has to be multiplied by 4. Table 2 shows the CC 0815 instruction set.

6.3 Encoding

When code needs to be generated the parser calls one of the following methods, which fill these formats into a byte buffer. When parsing is finished and all links are fixed, the buffer content is written to a file.

Listing 8: Encoding different formats

```
public void putF1(int opCode, int rx, int ry, int c);
public void putF2(int opCode, int rx, int ry, int rz);
public void putF3(int opCode, int c);
```

6.4 Decoding

Depending on the op code, the target machine decodes the single 32 bit instruction with these methods:

Listing 9: Decoding different formats

```
public void decodeF1();
public void decodeF2();
public void decodeF3();
```

Table 2: CC 0815 instruction set

OP number	instruction	description
1	ADDI	reg[a] = reg[b] + c
2	SUBI	reg[a] = reg[b] - c
3	MULI	reg[a] = reg[b] * c
4	DIVI	reg[a] = reg[b] / c
5	MODI	reg[a] = reg[b] % c
6	CMPI	reg[a] = reg[b] - c
7	LDW	reg[a] = mem[(reg[b] + c)/4]
8	STW	mem[(reg[b] + c)/4] = reg[a]
9	POP	reg[a] = mem[reg[b] / 4]; reg[b] = reg[b] + c
10	PSH	reg[b] = reg[b] - c; mem[reg[b] / 4] = reg[a]
11	BEQ	if (reg[a] == 0) PC += c * 4
12	BGE	if (reg[a] >= 0) PC += c * 4
13	BGT	if (reg[a] > 0) PC += c * 4
14	BLE	if (reg[a] <= 0) PC += c * 4
15	BLT	if (reg[a] < 0) PC += c * 4
16	BNE	if (reg[a] != 0) PC += c * 4
17	BR	PC += c * 4
18	BSR	reg[31] = PC + 4
19	STR	mem[(reg[b] + reg[c]) / 4] = reg[a]
20	ADD	reg[a] = reg[b] + reg[c]
21	SUB	reg[a] = reg[b] - reg[c]
22	MUL	reg[a] = reg[b] * reg[c]
23	DIV	reg[a] = reg[b] / reg[c]
24	MOD	reg[a] = reg[b] % reg[c]
25	CMP	reg[a] = reg[b] - reg[c]
26	RET	PC = reg[c]
27	FLO	open file
28	FLC	close file
29	RDC	read char
30	WRC	write char
31	LD	reg[a] = mem[(reg[b] + reg[c]) / 4]
32	JSR	reg[31] = PC + 4
33	PRC	print character
34	PRN	print number
35	PRV	print variable
36	PRR	print value of register

7 Symbol table

The symbol table contains all declared types, methods and variables. The two data structures that we used for our symbol table are shown in the next listing.

Listing 10: Used data structures for the symbol table

```
class SymbolNode{
  String name;
  int offset;
  int size;
  int nodeClass;
  Type type;
  String scope;
  SymbolNode params;
  SymbolNode next;
}

class Type {
  int form;
  int length;
  SymbolNode field;
  Type base;
}
```

SymbolNode: This record type is used to save information about the given variables, data types or methods.

- String name: the name of the variable, method or data type.
- int offset: the offset were the variable is stored in memory.
- int size: the size in bytes, if its a record type or an array variable.
- nodeClass: shows if the node represents a variable, data type or method.
- Type type: information about the variables data type.
- String scope: contains procedureContext or "global", if global.
- SymbolNode Params: the parameters if the nodeClass is method.
- SymbolNode next: the next SymbolNode in the table.

Type: This record type is used to save the types of variables, data types or methods.

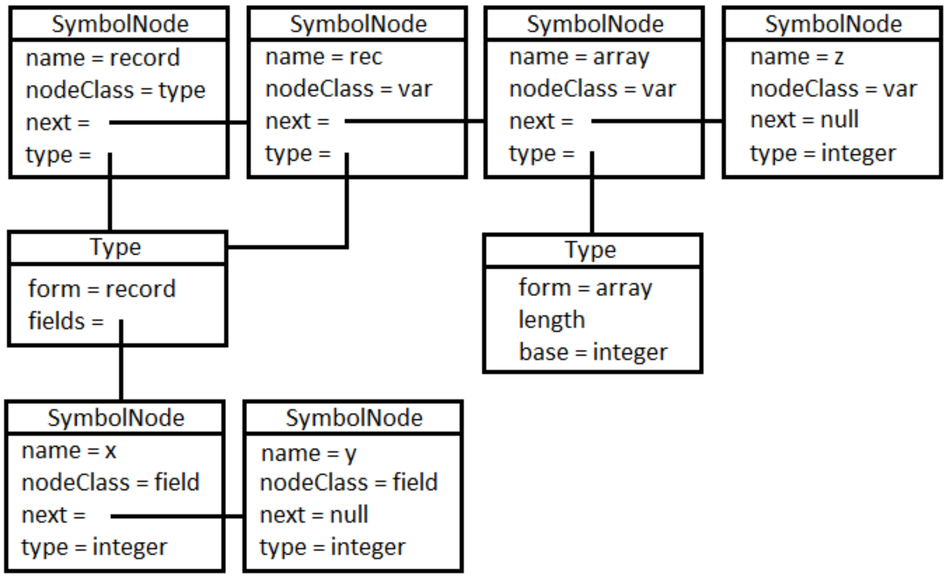
- form: identifies which type is represented.
- length: the length of an array.
- SymbolNode field: the fields of a record.
- Type base: the type of array fields.

As an example how the structure of our symbol table looks like see the code and graphic below.

Listing 11: Declaration example

```
class record{
  int x;
  int y;
}
int [] array;
int z;
record rec;
```

Figure 1: Symbol table of Listing 11



8 Arithmetic and boolean expressions

8.1 Arithmetic expressions

Our compiler is able to create code for arithmetic expressions, including the rules of commutativity and associativity. The supported operators are (,), +, -,* and /. We also support constant folding, which we will see in the following listening.

Listing 12: Constant folding

```java
public class ConstantFolding{
  int x;
  int y;
  int z;
  public static void main(String [] args){
    x = 1;
    y = 2;
    z = x + y;
    z = (4*3-4+2)/6;
  }
}
```

Listing 13: Generated instructions

```
 1 ADDI: R1  / R0  / 1         // x = 1;
 2 STW:  R1  / R28 / 0         // x = 1;
 3 ADDI: R2  / R0  / 2         // y = 2;
 4 STW:  R2  / R28 / -4        // y = 2;
 5 LDW:  R1  / R28 / 0         // z = x + y;
 6 LDW:  R2  / R28 / -4        // z = x + y;
 7 ADD:  R1  / R1  / R2        // z = x + y;
 8 STW:  R1  / R28 / -8        // z = x + y;
 9 ADDI: R1  / R0  / 1         // z = (4*3-4+2)/6;
10 STW:  R1  / R28 / -8        // z = (4*3-4+2)/6;
```

Let's have a look at the generated instructions. In the first four rows there is the value assignment for x and y. As you can see in line five to eight, a simple expression cause the compiler to generate four lines of code. This is because of the variables. Their values need to be loaded from the memory before we can compute them. In the last two lines we see now what constant folding does. We now know the values of the expression in compile time so we can compute them before code is generated. For that we do not need to generate extra instructions to calculate the expression.

8.2 Boolean expressions

Boolean expressions are also a feature our compiler supports. To increase performance we use lazy evaluation strategy. We will show this feature in the next section.

8.3 Lazy Evaluation

In order to show this feature, let's look at the following demo code:

Listing 14: Lazy evaluation

```
public class DemoLazyEvaluation {

  public boolean checkAnotherBoolean(boolean
     anotherBoolean) {
    System.out.println("--> called checkAnotherBoolean"
       );
    return anotherBoolean;
  }

  public static void main(String args) {
    boolean exit;
    int count;
    count = 100;
    exit = false;

    // From 100 to 91, there is no call of
       checkAnotherBoolean, because !exit
    // evaluates to true, so the whole OR expression
    // evaluates to true. If count == 90, the left hand
        side of the
    // expression evaluates to false, so there is
    // a call of checkAnotherBoolean, which also evals
       to false and the loop
    // condition evals to false.
    while (!exit || checkAnotherBoolean(!exit)) {
      count = count - 1;
      if (count == 90)
        exit = !exit;
      System.out.println("Count: " + count);
    }
  }
}
```

Listing 15: Execution of DemoLazyEvaluation

```
Count : 99
Count : 98
Count : 97
Count : 96
Count : 95
Count : 94
Count : 93
Count : 92
Count : 91
Count : 90
- - > called checkAnotherBoolean
```

9 Procedures

The support of procedures is essential, no one writes code in one single main procedure. Our compiler supports:

- procedures with or without parameters
- procedures with or without return value

Section 10 shows a demo, which contains different procedures.

9.1 Local hiding

This feature is used, when a variable occurs, which already exists in an outer context. Referring to the variable name will use the one, which is the closest one in scope, the one in the outer scope is *shadowed* or *hidden*. As an example:

Listing 16: Local hiding

```
int x;   //global
public int getSquare (int x){
  return x*x;
}
```

Every time a procedure body gets parsed, a seperate inner symbol table is built. In this case, this means that a local variable with the name 'x' is searched. If there is a match in the local symbol table (like in this case), the local variable is used, otherwise, there is a lookup in the global symbol table.

10 Demo

During this project we created some demo programs (which can be found in our repo) to test our compiler. In this section we want to demonstrate the "Sieve of Eratosthenes". This program shows important features our compiler supports, including records (classes), nested loops and conditionals, arrays, procedures and system output. The program calculates all primes up to 50, using a boolean array which represents "prime" or "not prime". *initialize(int)* is the setup procedure, which allocates the record and the array. *determineMultiples()* marks all multiples of prime numbers in a nested loop. *calcNumOfPrimes()* returns the total number of primes calculated. At the end, *outputResult(boolean)* prints all primes or non primes, this depends on the boolean parameter.
Listing 17 shows the program, listing 18 shows the generated output.

10.1 Sieve of Eratosthenes

Listing 17: Sieve of Eratosthenes

```
public class SieveOfEratosthenes {

class PrimeMap {
  boolean[] notPrime;
  int max;
  int countPrimes;
}

PrimeMap map;

public void initialize(int maxValue) {
  int i;

  System.out.println("Determining primes up to " +
      maxValue);
  map = new PrimeMap();
  map.max = maxValue;
  map.notPrime = new boolean[maxValue];
  i = 2;
  while (i < map.max) {
    map.notPrime[i] = false;
    i = i + 1;
  }
}

public void determineMultiples() {
  int i;
  int j;
  int helper;

  i = 2;
  while (i < map.max) {
    if (map.notPrime[i] == false) {
      helper = 2 * i;
      j = 3;
      while (helper < map.max) {
        map.notPrime[helper] = true;
        helper = j * i;
        j = j + 1;
      }
    }
    i = i + 1;
  }
}

public int calcNumOfPrimes() {
  int i;
  int count;

  count = 0;
  i = 2;
  while (i < map.max) {
    if (map.notPrime[i] == false)
      count = count + 1;
    i = i + 1;
  }
  return count;
}

public void outputResult(boolean requestPrimes) {
  int i;

  if (!requestPrimes) {
    System.out.println("ALL NON-PRIMES:");
    System.out.println("1");
  } else {
    System.out.println("ALL PRIMES:");
  }
  i = 2;
  while (i < map.max) {
    if (map.notPrime[i] != requestPrimes) {
      System.out.println(i);
    }
    i = i + 1;
  }
}

public static void main(String[] args) {
  System.out.println("
      ------------------------------------------------");
  System.out.println("COMPILER CONSTRUCTION SS 2013");
  System.out.println("Alexander Gruschina - Mario Kotoy
      ");
  System.out.println("CC0815 Demo");
  System.out.println("The Sieve of Eratosthenes");
  System.out.println("
      ------------------------------------------------");

  initialize(50);
  determineMultiples();
  map.countPrimes = calcNumOfPrimes();
  System.out.println("Total number of primes: " + map.
      countPrimes);
  outputResult(true);
  System.out.println("--------------- Done ---------------");
}
}
```

Listing 18: Execution of Sieve of Eratosthenes

```
------------------------------------------------
COMPILER CONSTRUCTION SS2013
Alexander Gruschina-Mario Kotoy
CC0815 Demo
The Sieve of Eratosthenes
------------------------------------------------
Determining primes up to 50
Total number of primes:15
ALL PRIMES:
2
3
5
7
11
13
17
```

```
19
23
29
31
37
41
43
47
———————————Done———————————
```

11 Team members

Alexander Gruschina	Mario Kotoy
agruschi@cosy.sbg.ac.at	mkotoy@cosy.sbg.ac.at

References

[1] Prof. Christoph Kirsch. Introduction to compiler construction, 2013.