Unidade Curricular de Compiladores

Mestrado Integrado em Engenharia Informática e Computação (MIEIC)

Departamento de Engenharia Informática

Universidade do Porto/FEUP

2º Semestre de 2010/2011

Compilador da linguagem yal0.2 para Bytecodes Java

João M. P. Cardoso and Rui Maranhão

Índice

Objectivo: Aplicar os conhecimentos adquiridos em compiladores considerando a construção de um compilador de programas descritos na linguagem *yal*. O compilador deve gerar

instruções JVM (*Java Virtual Machine*) aceites pelo *jasmin*, uma ferramenta de geração de bytecodes Java a partir de programas *assembly* com instruções JVM.

---

1. Introdução e Notas sobre a Avaliação

Pretende-se com este projecto que adquirem e apliquem conhecimentos no âmbito da UC (unidade curricular) de compiladores (3º ano do MIEIC). Para tal vão aprender a desenvolver um mini-compilador. O projecto encontra-se dividido em etapas que fundamentalmente correspondem a etapas no fluxo de compilação de um compilador real. É natural e expectável que neste projecto sintam alguns dos problemas que teriam no desenvolvimento de um compilador comercial. As etapas do projecto permitem que possam gerir os esforços e dedicação ao projecto conforme o estágio do mesmo que queiram atingir.

O projecto tem um peso de 70% (até 14 valores) na nota final da disciplina e os restantes 30% são atribuídos aos dois testes (ou exame de recurso). A nota obtida no projecto é distribuída pelos seguintes parâmetros:

i. 10% dedicados à organização e clareza do código desenvolvido, incluindo a documentação;
ii. 10% dedicados às soluções implementadas para resolver os problemas encontrados;
iii. 10% dedicados ao nível de optimizações que o compilador realiza;
iv. 70% dedicados aos resultados obtidos pelos testes funcionais realizados com duas baterias de exemplos de teste:
    a. 45% são obtidos pela avaliação do compilador com a bateria de testes previamente disponibilizada aos alunos;
    b. 25% são obtidos pela avaliação do compilador com a bateria de testes que não será previamente disponibilizada.

As notas dos membros de cada grupo serão tipicamente iguais. Notas diferentes indicam possíveis problemas, nomeadamente na distribuição do trabalho, no empenho, e nos esclarecimentos dados aos docentes.

Datas importantes:

- **checkpoint 1**: semana de 21 a 25 de Março (durante as aulas TPs)
- **checkpoint 2**: semana de 9 a 13 de Maio (durante as aulas TPs)
- **entrega do projecto**: 6 de Junho (submissão na página da disciplina no Moodle)

2. O Compilador yal2jvm

O compilador, designado por **yal2jvm**, deve traduzir programas em **yal**[1] versão 0.2 (a linguagem é introduzida na secção seguinte) para *bytecodes* Java [1]. A Figura 1 ilustra o fluxo de compilação pretendido. O compilador deve gerar ficheiros descritivos das classes com instruções da JVM aceites pelo *jasmin* [2], uma ferramenta que traduz essas classes em *bytecodes* Java (*classfiles*).

As classes geradas a partir de código **yal** podem ser integradas numa aplicação Java. Essas classes podem fazer referência a métodos classes Java previamente compiladas para *bytecodes* (mais à frente veremos a utilidade deste conceito).

O compilador **yal2jvm** deve ser executado utilizando:

java yal2jvm [-r=<num>] [-o] <input_file.yal>

ou

java –jar yal2jvm.jar [-r=<num>] [-o] <input_file.yal>

Em que *<input_file.yal>* representa o ficheiro que integra o módulo **yal** que se pretende compilar.

A opção "–r" indica ao compilador para usar apenas as primeiras *<num>*[2] variáveis locais da JVM. Sem a utilização da opção "–r" (que equivale à invocação do compilador com –r=0), as variáveis utilizadas nas funções do programa **yal** são armazenadas nas variáveis locais da JVM.

Com a opção "–o", o compilador deve realizar um conjunto de optimizações de código.

A secção 4 (página 6) descreve detalhadamente as opções "-r" e "-o".

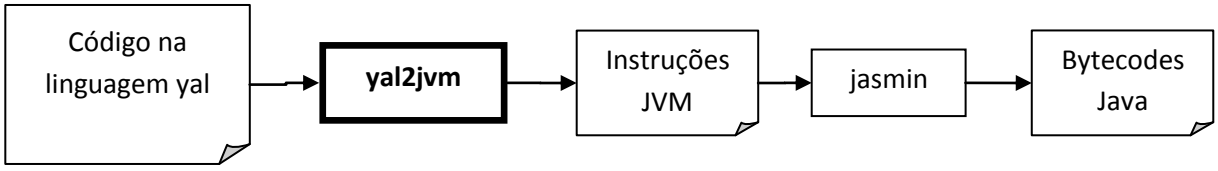

Figura 1. Fluxo de compilação pretendido.

O compilador deve gerar uma classe com o nome *<input_file>.j*. As classes *.j* são depois traduzidas para classes de *bytecodes* Java (*classfiles*) pela ferramenta *jasmin* [2].

[1] Não existe qualquer relação com a linguagem de programação YAL (*Yet Another Language*).
[2] <num> representa um número inteiro de 0 a 255.

O compilador deve contemplar as etapas de análise lexical, análise sintáctica, análise semântica, e de geração de código. A etapa de optimização (invocada pela opção "-o") é opcional.

Tratamento de Erros

Uma parte importante de qualquer compilador é a indicação dos erros associados a cada etapa de análise. Estes devem ser legíveis e úteis. Por exemplo, na análise sintáctica, o compilador deve reportar erros mais informativos do que os reportados por omissão pelo JavaCC. Aconselha-se a consulta do documento sobre tratamento e recuperação de erros no JavaCC [3]. Uma das possibilidades é fazer com que a análise desprezar os símbolos terminais (*tokens*) associados ao enunciado em que houve um erro. Por exemplo, no caso de um erro na expressão de um *while* a análise pode desprezar todos os *tokens* até ao primeiro "{".

A recuperação de erros permite evitar que determinados erros não originem erros desnecessários (que podem deixar de o ser depois de se corrigir o erro que lhes deu origem). Notem também que o compilador não deve terminar a execução logo após o primeiro erro que encontrar e deve reportar um conjunto de erros para que o programador proceda com as correcções dos mesmos (um compilador pode reportar os 10 primeiros erros que encontrar, por exemplo).

Análise Semântica

De forma a reduzir o trabalho relacionado com a análise semântica, o compilador a implementar deve apenas realizar a análise semântica relacionada com a invocação de funções cujo código esteja contido no módulo que se pretende compilar. Para a invocação de outras funções (i.e., funções de outros módulos ou em *classfiles*) deve assumir-se que não existem erros semânticos.

Os erros semânticos detectados devem ser reportados e em caso de erros a execução do compilador deve terminar na etapa de análise semântica.

Representação Intermédia

A geração de código pode ter como ponto de partida a AST[3] e a(s) tabela(s) de símbolos que representam o módulo **yal**. No entanto, um compilador mais sofisticado pode usar nas etapas de optimizações e de geração de código uma representação baseada em código de três endereços[4], por exemplo.

[3] Do Inglês: *Abstract Syntax Tree* (árvore de sintaxe abstracta).
[4] Do Inglês: *Three address code*.

3. A linguagem yal

Um programa na linguagem **yal** é constituído por um ou mais módulos. Cada ficheiro de programa tem um módulo **yal**. Cada módulo **yal** tem um nome, pode ter atributos, e funções. A Figura 2 ilustra o primeiro programa em **yal**. A função *main()* corresponde ao método principal do módulo[5]. A gramática em BNF (*Backus–Naur Form*) da linguagem **yal** encontra-se no *Apêndice A: Gramática da linguagem yal em BNF* (ver página 11).

```
module primeiro_programa {
   function main() {
      io.print ("Hello World");
   }
}
```

Figura 2. Primeiro programa em yal.

A linguagem **yal** não contempla suporte especifico a primitivas de leitura ou de escrita (e.g., do teclado ou de ficheiro). Essas operações podem ser realizadas pelo uso de *classfiles* que integram funções como *read()*, *print(...)* e *println(...)*. No âmbito da validação do compilador a desenvolver, estas funções estão definidas na classe **io** fornecida (ver o *Apêndice B: O módulo io*, página 11).

Algumas características da linguagem **yal**:

- Em **yal** existem apenas variáveis escalares do tipo inteiro e *arrays* unidimensionais com elementos do tipo inteiro. Todos os dados são por omissão do tipo inteiro (nesta versão da linguagem não existem outros tipos de dados);
- As variáveis escalares não podem atingir possíveis usos sem terem tido valores atribuídos (como em Java);
- Os *arrays* têm de ser criados antes de serem acedidos (e.g., a instrução *A=[20];* declara um *array* com 20 elementos do tipo inteiro);
- Em cada função **yal**, um identificador só pode ser usado para identificar uma variável escalar ou uma variável do tipo *array* (por exemplo, na mesma função o mesmo nome não pode ser usado para identificar um escalar e depois um *array*);
- Podem ser usados os mesmos identificadores para representar nomes de funções e de variáveis (i.e., num módulo **yal** podem co-existir uma função designada por "max" e uma variável designada por "max");
- Na passagem de argumentos e devolução de resultados, as variáveis escalares são passadas por valor e as variáveis do tipo *array* são passadas por referência;
- Existe apenas um tipo de *loop*: o *while*;
- Cada módulo **yal** pode ser visto como uma classe estática em Java, com atributos e métodos *static*;

[5] esta função deverá ser implementada pelo método main do Java: "public static void main(String [] args)"

- A linguagem não distingue entre maiúsculas e minúsculas (i.e., é *case insensitive*).

A Figura 3 apresenta um programa em **yal** que utiliza dois módulos: um módulo principal e um módulo onde estão definidas duas funções utilizadas no módulo principal. Notem que este programa supõe a existência da classe ***io*** com o método equivalente à função invocada no código **yal** (i.e., *io.println(...)*).

ficheiro programa1.yal

```
module programa1 {

    data=[100]; // vector of 100 integers
    mx; // attribute mx
    mn; // attribute mn

    function det(d[]) { // N = size(d)
        i=0;
        M=d.size-1; // d.size equivales to d.length (Java)
        while(i<M) { // version not optimized!
          a=d[i]; i=i+1; b=d[i];
          mx= library1.max(a,b);
          mn= library1.min(a,b);
        }
    }

    function main() {
        det(data);
        io.println("max: ",mx);
        io.println("min: ",mn);
    }
}
```

(a) módulo principal;

ficheiro library1.yal

```
module library1 {

    function m=max(a,b) {
        if(a > b) {
            m = a;
        } else {
            m = b;
        }
    }

    function m=min(a,b) {
        if(a > b) {
            m = b;
        } else {
            m = a;
        }
    }

}
```

(b) módulo com funções auxiliares;

Figura 3. Segundo programa em yal.

4. Optimizações e Alocação de Registos no Compilador yal2jvm

Estas etapas são necessárias para grupos que queiram atingir notas de projecto de 18 a 20 valores. Espera-se que por omissão (i.e., sem necessidade da opção "–o"), o compilador gere código JVM [1] com instruções de menor custo nos casos: iload, istore, astore, aload, carregamento de constantes na pilha, utilização de iinc, etc.

Todas as optimizações que forem incluídas no vosso compilador devem ser identificadas no ficheiro **README.txt** que devem entregar aquando da submissão do projecto.

Opção –r=<n>

Com a opção "–r", o compilador tenta afectar as variáveis utilizadas nas funções do programa **yal** às <*num*> primeiras variáveis locais da JVM:

- Deve reportar as variável utilizadas em cada função do programa **yal** que são afectas a cada variável local da JVM.
- Se o valor de *n* não for suficiente para armazenar as variáveis na função então o compilador deve abortar a execução, reportar um erro e indicar o número de variáveis locais da JVM necessário.
- Esta opção requer o cálculo do tempo de vida das variáveis utilizando análise do fluxo de dados (*dataflow analysis*). Vejam os slides das aulas teóricas. Uma implementação eficiente na análise de fluxo de dados pode utilizar para os conjuntos *def* e *use* objectos da classe BitSet do Java.
- A "alocação de registos" (afectação das variáveis locais da função a *n* variáveis locais da JVM, neste caso) poderia ser realizada com o uso do algoritmo de coloração de grafos explicado nas aulas teóricas. Contudo, e devido ao facto de poderem não ter tempo suficiente para dedicar a esta fase, aceita-se que utilizem outro algoritmo para a "alocação de registos".

Opção "–o":

Pretende-se que a opção "-o" integre as duas optimizações seguintes:

- Identificar usos de variáveis locais a funções que possam ser substituídos por constantes (permite que não seja necessária a afectação de variáveis locais da JVM para armazenar o valor de variáveis apenas usadas para armazenar valores estaticamente conhecidos). Notem que nesta optimização não necessitam de inclui a avaliação de expressões com constantes.
- Utilizar templates para os "while" que evitam a utilização de um "goto" logo a seguir ao salto condicional que controla a execução de uma nova iteração do "loop" ou do término do mesmo (quem tiver incluído esta optimização na versão não optimizante do compilador não necessita de fazer alterações)

E uma outra optimização do código ao critério do grupo. Surpreendam-nos!!!!

5. Etapas Sugeridas para o Compilador

Etapas sugeridas para o desenvolvimento do compilador **yal2jvm**:

1. Desenvolva o *parser* de **yal** em JavaCC tomando como ponto de partida a gramática disponibilizada (note que a gramática pode originar conflitos quando implementada com

parsers do tipo LL(1)[6] e por isso deve modificar a gramática de forma a eliminar esses conflitos);

2. Proceda à especificação do ficheiro **jjt**[7] para gerar, utilizando o JJTree, uma nova versão do *parser* que inclua a geração da árvore (a árvore gerada deve ser uma AST[8]), anotando os nós e folhas da árvore com a informação necessária para realizar os passos de compilação seguintes;
3. Inclua a construção das tabelas de símbolos necessárias;
4. Análise Semântica[9]; **[*checkpoint*[10] 1]**
5. Gere código JVM aceite pelo *jasmin* correspondente a invocação de funções em **yal**;
6. Gere código JVM aceite pelo *jasmin* para expressões aritméticas;
7. Gere código JVM aceite pelo *jasmin* para instruções condicionais (*if* e *if-else*);
8. Gere código JVM aceite pelo *jasmin* para loops; **[*checkpoint* 2]**
9. Gere código JVM aceite pelo *jasmin* para vectores.
10. Complete o compilador e teste-o com um conjunto de exemplos;
11. Proceda a optimizações na geração de código, relacionadas nomeadamente com a afectação de variáveis escalares locais a funções a variáveis locais da JVM (opção "-r") e a optimizações relacionadas com a opção "-o".
    [esta tarefa é necessária para alunos que pretendam ter nota do projecto igual ou superior a 18 valores]

Assim que tenham a etapa de geração da AST concluída, é boa ideia procederem às etapas do compilador necessárias para fornecerem código da JVM para o exemplo da Figura 2. Podem depois generalizar para terem em conta invocação de funções existentes em módulos **yal** externos ou no próprio módulo.

Deve notar-se que o compilador deve reportar possíveis erros nas etapas de análise gramatical e de análise semântica.

6. Instruções JVM e Geração de Bytecodes Java

Como base inicial para a aprendizagem das instruções JVM e dos *bytecodes* Java gerados ilustra-se de seguida um conjunto de exemplos. A Figura 4 apresenta um programa Java simples que escreve no ecrã "Hello World".

[6] Embora seja aconselhado, não é estritamente necessário usar *lookahead* de valor 1.
[7] Basicamente pode copiar o ficheiro jj e fazer as alterações para que o JJTree gere o código necessário para gerar as árvores de análise.
[8] *Abstract Syntax Tree*.
[9] O compilador não verifica se invocações a funções cujo código não esteja incluído no ficheiro a compilar estão de acordo com o protótipo dessas funções.
[10] Corresponde à apresentação das etapas desenvolvidas.

```
public class Hello {
        public static void main(String args[]) {
                System.out.println("Hello World!");
        }
}
```

Figura 4. Classe "Hello" no ficheiro "Hello.java".

Depois de compilado com o *javac* (*javac Hello.java*) obtém-se o ficheiro *Hello.class* (ficheiro com os *bytecodes* Java para a classe especificada). Para visualizarmos as instruções JVM e os atributos da classe (ver Figura 5) poderemos executar[11]:

```
javap –c Hello
```

```
public class Hello extends java.lang.Object{
public Hello();
  Code:
   0:     aload_0
   1:     invokespecial  #1; //Method java/lang/Object."<init>":()V
   4:     return
public static void main(java.lang.String[]);
  Code:
   0:     getstatic        #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   3:     ldc      #3; //String Hello World!
   5:     invokevirtual  #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   8:     return
}
```

Figura 5. Instruções JVM nos *bytecodes* obtidos após compilação pelo *javac* da classe "Hello".

Podemos programar com instruções JVM uma classe equivalente à classe anterior. Neste caso vamos gerar os *bytecodes* Java a partir do *assembly* da JVM utilizando o *jasmin*. A classe equivalente à classe *Hello* (no ficheiro *Hello.java*) é a classe descrita no ficheiro *Hello.j* utilizando uma linguagem *assembly* com instruções da JVM aceites pelo *jasmin*. A Figura 6 ilustra o código da classe *Hello* no ficheiro *Hello.j*.

[11] Para obter informação sobre o número de variáveis locais e o número de níveis da pilha necessários para cada método pode usar-se: javap –verbose Hello.

```
; Classe com sintaxe aceite pelo Jasmin2.3

.class public Hello
.super java/lang/Object

;
; standard initializer
.method public <init>()V
  aload_0

  invokenonvirtual java/lang/Object/<init>()V
  return
.end method

.method public static main([Ljava/lang/String;)V
  .limit stack 2
  ;.limit locals 2 ; este exemplo não requer variáveis locais

  getstatic java/lang/System.out Ljava/io/PrintStream;
  ldc "Hello World!"
  invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
  return
.end method
```

Figura 6. Programação da classe "Hello" (ficheiro "Hello.j") utilizando instruções JVM e sintaxe aceite pelo *jasmin*.

Poderemos agora gerar os bytecodes Java para esta classe utilizando o *jasmin*:

java –jar jasmine.jar Hello.j

(obtém-se assim a *classfile Hello.class* que tem a funcionalidade da classe gerada anteriormente a partir do código Java)

7. Referências

[1] The Java Virtual Machine Specification, http://java.sun.com/docs/books/jvms/
[2] Jasmin Home Page, http://jasmin.sourceforge.net/
[3] JavaCC [tm]: Error Reporting and Recovery, http://javacc.java.net/doc/errorrecovery.html
[4] JavaCC, http://java.net/projects/javacc

Apêndice A: Gramática da linguagem yal em BNF

TOKENS

```
<DEFAULT> SKIP : {
" "
| "\t"
| "\n"
| "\r"
| <"//" (~["\n","\r"])* ("\n" | "\r" | "\r\n")>
| <"/*" (~["*"])* "*" ("*" | ~["*","/"] (~["*"])* "*")* "/">
}


/* reserved words */
<DEFAULT> TOKEN : {
<RELA_OP: ">" | "<" | "<=" | ">=" | "==" | "!=">
| <ADDSUB_OP: "+" | "-">
| <ARITH_OP: "*" | "/" | "<<" | ">>" | ">>>">
| <BITWISE_OP: "&" | "|" | "^">
| <NOT_OP: "!">
| <WHILE: "while">
| <IF: "if">
| <ELSE: "else">
| <ASSIGN: "=">
| <ASPA: "\"">
| <LPAR: "(">
| <RPAR: ")">
| <VIRG: ",">
| <PVIRG: ";">
| <LCHAVETA: "{">
| <RCHAVETA: "}">
| <FUNCTION: "function">
| <MODULE: "module">
| <SIZE: "size">
}


<DEFAULT> TOKEN : {
<INTEGER: (<DIGIT>)+>
| <ID: <LETTER> (<LETTER> | <DIGIT>)*>
| <#LETTER: ["$","A"-"Z","_","a"-"z"]>
| <#DIGIT: ["0"-"9"]>
| <STRING: "\"" (["a"-"z","A"-"Z","0"-"9",":"," ","="])+ "\"">
}
```

NON-TERMINALS

Module ::= <MODULE> <ID> <LCHAVETA> (Declaration)* (Function)* <RCHAVETA>

Declaration ::= (ArrayElement | ScalarElement) (<ASSIGN> (("[" ArraySize "]") | (<ADDSUB_OP>)? <INTEGER>))? <PVIRG>

Function ::= ((<FUNCTION> (ArrayElement | ScalarElement) <ASSIGN> <ID> <LPAR> (Varlist)? <RPAR>) | (<FUNCTION> <ID> <LPAR> (Varlist)? <RPAR>)) <LCHAVETA> Stmtlst <RCHAVETA>

Varlist ::= (ArrayElement | ScalarElement) (<VIRG> (ArrayElement | ScalarElement))*

ArrayElement ::= <ID> "[" "]"

ScalarElement ::= <ID>

Stmtlst ::= (Stmt)*

Stmt ::= While

| If

| Assign

| Call <PVIRG>

Assign ::= Lhs <ASSIGN> Rhs <PVIRG>

Lhs ::= ArrayAccess

| ScalarAccess

Rhs ::= (Term ((<ARITH_OP> | <BITWISE_OP> | <ADDSUB_OP>) Term)?)

| "[" ArraySize "]"

ArraySize ::= ScalarAccess

| <INTEGER>

Term ::= (<ADDSUB_OP>)? (<INTEGER> | Call | ArrayAccess | ScalarAccess)

Exprtest ::= <LPAR> Lhs <RELA_OP> Rhs <RPAR>

While ::= <WHILE> Exprtest <LCHAVETA> Stmtlst <RCHAVETA>

If ::= <IF> Exprtest <LCHAVETA> Stmtlst <RCHAVETA> (<ELSE> <LCHAVETA> Stmtlst <RCHAVETA>)?

Call ::= <ID> ("." <ID>)? <LPAR> (ArgumentList)? <RPAR>

ArgumentList ::= Argument (<VIRG> Argument)*

Argument ::= (<ID> | <STRING> | <INTEGER>)

ArrayAccess ::= <ID> "[" Index "]"

ScalarAccess ::= <ID> ("." <SIZE>)?

Index ::= <ID>

| <INTEGER>

Apêndice B: O módulo io

Para teste do compilador vamos utilizar o módulo ***io*** disponibilizado sobre a forma de Java *bytecodes*:

io

int read(); // lê um inteiro introduzido pelo teclado
 	
void print(String, int); // escreve no ecrã a string seguida do valor do inteiro
 	
void print(String); // escreve no ecrã a string
 	
void println(String, int); // escreve no ecrã a string seguida do valor do inteiro e muda de linha
 	
void println(String); // escreve no ecrã a string e muda de linha
 	
void println(); // muda de linha

A classe Java equivalente (*io.class*) a esta biblioteca é disponibilizada para este projecto. Os protótipos dos métodos incluídos nessa classe são ilustrados em baixo.

publict static int read() publict static void print(String, int) publict static void print(String) publict static void print(int) publict static void println(String, int) publict static void println(String) publict static void println(int) publict static void println()