
UNIVERSIDADE DE COIMBRA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
*Departamento de Engenharia Informática*

**Projeto de Compiladores**

**2013/14 – 2º semestre**

Licenciatura em Engenharia Informática

**Entrega final:** 30 de Maio de 2014

v1.9.0

**Nota Importante: A fraude denota uma grave falta de ética e constitui um comportamento inadmissível num estudante do ensino superior e futuro profissional licenciado. Qualquer tentativa de fraude leva à reprovação à disciplina tanto do facilitador como do prevaricador.**

# Compiladores

## Compilador para a linguagem iJava

Este projeto consiste no desenvolvimento de um compilador para a linguagem iJava (imperative Java), que consiste num pequeno subconjunto da linguagem Java (versão 5.0). Os programas da linguagem iJava são constituídos por uma única classe (a classe principal), contendo necessariamente um método `main`, e podendo conter outros métodos e atributos, todos eles estáticos e (possivelmente) públicos.

Na linguagem iJava é possível utilizar variáveis e literais dos tipos inteiro (de 32 bits) com sinal e booleano, e variáveis dos tipos array de inteiros e array de valores booleanos (com uma única dimensão). A linguagem implementa expressões aritméticas e lógicas e operações relacionais simples, bem como instruções de atribuição e de controlo (`if-else` e `while`). Implementa ainda métodos (estáticos) envolvendo quaisquer dos tipos de dados referidos acima e/ou o tipo de retorno `void`.

É possível passar parâmetros, que deverão ser literais inteiros, a um programa iJava através da linha de comandos. Supondo que o nome dado ao argumento do método `main()` é `args`, os seus valores podem ser recuperados através da construção `Integer.parseInt(args[...])`, e o número de parâmetros pode ser obtido através da expressão `args.length`. A construção `System.out.println(...)` permite imprimir valores inteiros ou lógicos.

Finalmente, são aceites (e ignorados) comentários dos tipos `/* ... */` e `//...`.

O significado de um programa em iJava será o mesmo que o seu significado em Java. Por exemplo, o seguinte programa imprime o primeiro argumento passado na linha de comandos:

```
class echo {
     public static void main(String[] args) {
          int x;
          x = Integer.parseInt(args[0]);
          System.out.println(x);
     }
}
```

## Fases

O projeto será estruturado como uma sequência de três metas, a saber:

1. Análise lexical (10%) – 21 de março de 2014
2. Análise sintática, construção da árvore de sintaxe abstrata, análise semântica (tabelas de símbolos, deteção de erros semânticos) (55%) – 28 de abril de 2014
3. Geração de código (20%) + relatório (15%) – 30 de maio de 2014

Em cada uma das metas, o trabalho deverá ser validado no mooshak, usando um concurso criado especificamente para o efeito. Para além disso, a entrega final do projeto deverá ser feita no inforestudante até às **23h59** do dia **30 de Maio**, e incluir o relatório e todo o software criado.

## Defesa e grupos

O trabalho será normalmente realizado por grupos de dois alunos, admitindo-se também que o seja a título individual. A defesa oral do trabalho terá lugar na primeira semana do mês de junho. A nota da defesa (entre 0 e 100%) multiplica pela média ponderada das pontuações obtidas no mooshak e no relatório à data de cada uma das metas. *Excecionalmente*, e por motivos justificados (como, por exemplo, falha técnica), poderão ser atribuídas notas superiores a 100% na defesa, mas a classificação final nunca poderá exceder a pontuação obtida no mooshak para as diversas fases à data da última entrega.

Aplicam-se mínimos de 47,5% à nota final após a defesa.

## Bibliografia


. Rui Gustavo Crespo, *Processadores de Linguagensm* IST Press, 1998

. A. Appel, *Modern compiler implementation in C*. Cambridge Press, 1998.

. T. Niemann, *A Compact Guide to Lex & Yacc,*
http://epaperpress.com/lexandyacc/epaperpress.

. Manual do yacc em Unix (comando "man yacc" na shell)

. John R. Levine, Tony Mason and Doug Brown, *Lex & Yacc,* O'Reilly. 2004

. Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, 2nd edition, 1988.

## *Fase I – Analisador lexical*

O analisador lexical deve ser implementado em C utilizando a ferramenta `lex`. Os tokens da linguagem são apresentados de seguida.

## Tokens da linguagem iJava

ID: Sequências alfanuméricas começadas por uma letra, onde os símbolos "_" e "$" contam como letras. Maiúsculas e minúsculas são consideradas letras diferentes.

INTLIT: Sequências de dígitos decimais, e sequências de dígitos hexadecimais (incluindo a-f e A-F) precedidas de "0x".

BOOLLIT = "true" | "false"
INT = "int"
BOOL = "boolean"
NEW = "new"
IF = "if"
ELSE = "else"
WHILE = "while"
PRINT = "System.out.println"
PARSEINT = "Integer.parseInt"
CLASS = "class"
PUBLIC = "public"
STATIC = "static"
VOID = "void"
STRING = "String"
DOTLENGTH = ".length"
RETURN = "return"
OCURV = "("
CCURV = ")"
OBRACE = "{"
CBRACE = "}"
OSQUARE = "["
CSQUARE = "]"
OP1 = "&&" | "||"
OP2 = "<" | ">" | "==" | "!=" | "<=" | ">="
OP3 = "+" | "-"
OP4 = "*" | "/" | "%"
NOT = "!"
ASSIGN = "="
SEMIC = ";"
COMMA = ","
RESERVED = keywords do Java não utilizadas em iJava, bem como o literal "null".

## Implementação

O analisador deverá chamar-se `ijscanner`, ler o ficheiro a processar através do `stdin`, e emitir o resultado da análise para o `stdout`. Caso o ficheiro `echo.ijava` contenha o programa de exemplo dado anteriormente, a invocação

```
./ijscanner < echo.ijava
```

deverá imprimir a correspondente sequência de tokens no ecrã. Neste caso:

```
CLASS
ID(echo)
OBRACE
PUBLIC
STATIC
VOID
ID(main)
OCURV
STRING
OSQUARE
CSQUARE
ID(args)
CCURV
OBRACE
INT
ID(x)
SEMIC
ID(x)
ASSIGN
PARSEINT
OCURV
ID(args)
OSQUARE
INTLIT(0)
CSQUARE
CCURV
SEMIC
PRINT
OCURV
ID(x)
CCURV
SEMIC
CBRACE
CBRACE
```

O analisador deve aceitar (e ignorar) comentários dos tipos `/* ... */` e `//...`, e detetar a existência de quaisquer erros lexicais no ficheiro de entrada. Sempre que um token possa admitir mais do que um valor semântico, o valor encontrado deve ser impresso entre parêntesis logo a seguir ao nome do token, como se exemplificou acima para `ID` e `INTLIT`.

## Tratamento de erros

Caso o ficheiro de entrada contenha erros lexicais, o programa deverá imprimir uma das seguintes mensagens no `stdout`, conforme o caso:

- "Line <num linha>, col <num coluna>: illegal character ('<c>')\n"
- "Line <num linha>, col <num coluna>: unterminated comment\n"

onde <c>, <num linha> e <num coluna> devem ser substituídos pelos valores correspondentes ao (início do) token que originou o erro. O analisador deve recuperar da ocorrência de erros lexicais a partir do fim desse token.

## Submissão

O trabalho deverá ser validado no mooshak, usando o concurso criado especificamente para o efeito em `http://mooshak.dei.uc.pt/~comp2014`. Será tida em conta apenas a última submissão ao problema A desse concurso. Os restantes problemas destinam-se a ajudar na validação do analisador. No entanto, o mooshak não deve ser utilizado como ferramenta de debug!

O ficheiro lex a submeter deve chamar-se `ijscanner.l` e ser colocado num ficheiro zip com o nome `ijscanner.zip`. O ficheiro zip não deve conter quaisquer diretórios.

## *Fase II – Analisador sintático e semântico*

O analisador sintático deve ser implementado em C utilizando as ferramentas `lex` e `yacc`. A gramática seguinte define a sintaxe da linguagem iJava.

## Gramática inicial em notação EBNF

```
Start → Program
Program → CLASS ID OBRACE { FieldDecl | MethodDecl } CBRACE
FieldDecl → STATIC VarDecl
MethodDecl → PUBLIC STATIC ( Type | VOID ) ID OCURV
  [ FormalParams ] CCURV OBRACE { VarDecl } { Statement } CBRACE
FormalParams → Type ID { COMMA Type ID }
FormalParams → STRING OSQUARE CSQUARE ID
VarDecl → Type ID { COMMA ID } SEMIC
Type → ( INT | BOOL ) [ OSQUARE CSQUARE ]
Statement → OBRACE { Statement } CBRACE
Statement → IF OCURV Expr CCURV Statement [ ELSE Statement ]
Statement → WHILE OCURV Expr CCURV Statement
Statement → PRINT OCURV Expr CCURV SEMIC
Statement → ID [ OSQUARE Expr CSQUARE ] ASSIGN Expr SEMIC
Statement → RETURN [ Expr ] SEMIC
Expr → Expr ( OP1 | OP2 | OP3 | OP4 ) Expr
Expr → Expr OSQUARE Expr CSQUARE
Expr → ID | INTLIT | BOOLLIT
Expr → NEW ( INT | BOOL ) OSQUARE Expr CSQUARE
Expr → OCURV Expr CCURV
Expr → Expr DOTLENGTH | ( OP3 | NOT ) Expr
Expr → PARSEINT OCURV ID OSQUARE Expr CSQUARE CCURV
Expr → ID OCURV [ Args ] CCURV
Args → Expr { COMMA Expr }
```

Uma vez que a gramática dada é ambígua e é apresentada em notação EBNF, onde […] representa "opcional" e {…} representa "zero ou mais repetições," esta deverá ser modificada para permitir a análise sintática ascendente com o `yacc`. Será necessário ter em conta a precedência e as regras de associação dos operadores, entre outros aspetos, de modo a garantir a compatibilidade entre as linguagens iJava e Java. Em particular, uma vez que os operadores "--" e "++" não estão disponíveis em iJava, deverão ser acrescentados à categoria lexical RESERVED para não serem entendidos como dois operadores unários seguidos. Além disso, como não existem arrays multi-dimensionais em iJava, expressões como:

```
new int[5][2]
```

devem dar origem a um erro sintático em vez de serem tratadas como a indexação de um array de dimensão 5 pelo valor 2, o que seria semanticamente válido, mas incompatível com o Java.

O analisador deverá chamar-se `ijparser`, ler o ficheiro a processar através do `stdin`, e emitir o resultado para o `stdout`. Caso o ficheiro `gcd.ijava` contenha o programa

```
class gcd {
  public static void main(String[] args) {
    int[] x;
    x = new int[2];
    x[0] = Integer.parseInt(args[0]);
    x[1] = Integer.parseInt(args[1]);
    if (x[0] == 0)
      System.out.println(x[1]);
    else
      while (x[1] > 0) {
        if (x[0] > x[1])
          x[0] = x[0] - x[1];
        else
          x[1] = x[1] - x[0];
      }
    System.out.println(x[0]);
  }
}
```

a invocação

```
./ijparser -t < gcd.ijava
```

deverá gerar a árvore de sintaxe abstrata correspondente, imprimi-la no ecrã conforme a seguir se explica, e terminar sem proceder à análise semântica (descrita mais adiante).

## Árvore de sintaxe abstrata

As árvores de sintaxe abstrata geradas durante a análise sintática devem incluir apenas nós dos tipos enumerados abaixo. Entre parêntesis à frente de cada nó indica-se o número de filhos desse nó e, onde necessário, também o tipo de filhos.

**Nó raiz**

```
Program(>=1)          ( Id { VarDecl | MethodDecl} )
```

**Declaração de variáveis**

```
VarDecl(>=2)          ( <type> Id {Id} )
```

**Definição de métodos**

```
MethodDecl(4)         ( ( <type> | Void ) Id MethodParams MethodBody)
MethodParams(>=0)     ( { ParamDeclaration } )
MethodBody(>=0)       ( { VarDecl | <statements> } )
ParamDeclaration(2)   ( ( <type> | StringArray ) Id )
```

**Statements**

```
CompoundStat(>=2) IfElse(3) Print(1) Return(1) Store(2) StoreArray(2)
While(2)
```

**Operadores**

```
Or(2) And(2) Eq(2) Neq(2) Lt(2) Gt(2) Leq(2) Geq(2) Add(2) Sub(2)
Mul(2) Div(2) Mod(2) Not(1) Minus(1) Plus(1) Length(1) LoadArray(2)
Call(>=1) ParseArgs(1)
```

**Terminais**

```
Int Bool IntArray BoolArray StringArray Void Id IntLit BoolLit
```

**Especial**

```
Null        (a imprimir na ausência de um nó filho)
```

No caso do programa dado:

```
Program
           ...
```

O analisador deve aceitar (e ignorar) comentários dos tipos `/* ... */` e `//...`, e detetar a existência de quaisquer erros lexicais ou de sintaxe no ficheiro de entrada.

## Tratamento de erros

Caso o ficheiro de entrada contenha erros lexicais, o programa deverá imprimir no stdout as mensagens definidas na Meta I, e continuar. Caso seja encontrado um erro de sintaxe, o analisador deve imprimir uma mensagem de erro e parar. A mensagem a imprimir será

- "Line <num linha>, col <num coluna>: syntax error: <token>\n"

onde <num linha>, <num coluna> e <token> devem ser substituídos pelos números de linha e de coluna, e pelo valor semântico do token que dá origem ao erro. Isto pode ser conseguido definindo a função:

```
void yyerror (char *s) {
     printf ("Line %d, col %d: %s: %s\n", <num linha>, <num
coluna>, s, yytext);
}
```

A árvore de sintaxe abstrata só deverá ser impressa se não houver erros de sintaxe. Caso haja erros lexicais que não causem também erros de sintaxe, a árvore deverá ser impressa imediatamente a seguir às correspondentes mensagens de erro.

## Análise semântica

Supondo que o programa de entrada é sintaticamente válido, a invocação

```
./ijparser -s < gcd.ijava
```

deverá levar o analisador a proceder à análise semântica do programa e a imprimir no `stdout` a(s) tabela(s) de símbolos como a seguir se especifica. Caso sejam passadas ambas a opções (-t e -s), a árvore de sintaxe abstrata deve ser impressa primeiro, seguida da(s) tabela(s) de símbolos (sem qualquer linha em branco a separá-las).

## Tabelas de símbolos

As linhas das tabelas de símbolos a imprimir com a opção -s devem seguir o formato "`Name\tType[\tparam]`", como se ilustra para o programa de exemplo da meta 2.

```
===== Class gcd Symbol Table =====
main  method
```

```
===== Method main Symbol Table =====
return      void
args  String[]    param
x     int[]
```

Os símbolos (e as tabelas) devem ser apresentados por ordem de declaração no programa fonte. A string "`return`" é usada para representar o valor de retorno do métodos e o pseudo-tipo "`String[]`" é usado para indicar o identificador que é usado no programa para aceder aos parâmetros passados na linha de comandos. Deve ser deixada um linha em branco entre tabelas consecutivas.

## Tratamento de erros semânticos

… A completar…

## Desenvolvimento do analisador

Sugere-se que o desenvolvimento do analisador seja efetuado em quatro fases. A primeira deverá visar a tradução da gramática para o yacc de modo a permitir detetar eventuais erros de sintaxe. A segunda deverá incidir sobre a construção da árvore de sintaxe abstrata e sua impressão na saída. A terceira deverá consistir na construção das tabelas de símbolos e sua impressão, e a quarta na verificação de erros semânticos.

## Submissão

O trabalho deverá ser validado no mooshak, usando o concurso criado especificamente para o efeito em `http://mooshak.dei.uc.pt/~comp2014`. Como na fase anterior, será tida em conta apenas a última submissão ao problema A desse concurso. Os restantes problemas destinam-se a ajudar na validação do analisador, nomeadamente no que respeita à deteção de erros de sintaxe e à construção da árvore de sintaxe abstrata, de acordo com a estratégia de desenvolvimento proposta.
Os ficheiros lex e yacc a submeter devem chamar-se `ijparser.l` e `ijparser.y` e ser colocados juntamente com quaisquer ficheiros adicionais necessários à compilação do analisador num único ficheiro zip com o nome `ijparser.zip`. O ficheiro zip não deve conter quaisquer diretórios.