



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

Projeto de Compiladores

2016/17 – 2º semestre

Licenciatura em Engenharia Informática

Data de Entrega Final: 2 de Junho 2017

v2.0.2

Nota Importante: Qualquer tipo de fraude terá como consequência imediata a reprovação à disciplina e será comunicada superiormente nos termos da regulamentação aplicável.

Compilador para a linguagem Ja

Este projeto consiste no desenvolvimento de um compilador para a linguagem Ja, que é um pequeno subconjunto da linguagem Java (versão 8)¹. Os programas da linguagem Ja são constituídos por uma única classe (a classe principal) contendo métodos e atributos, todos eles estáticos e (possivelmente) públicos. O ponto de entrada no programa é o método `main`.

Na linguagem Ja é possível utilizar variáveis e literais dos tipos booleano, inteiro (de 32 bits) com sinal e real de precisão dupla. Também é possível usar literais do tipo cadeia de caracteres (string), mas apenas para efeitos de impressão no `stdout`. A linguagem implementa expressões aritméticas e lógicas e operações relacionais simples, bem como instruções de controlo (`if-else`, `while` e `do-while`). Implementa ainda métodos (estáticos) envolvendo os tipos de dados referidos acima (e ainda o tipo especial `String[]`), com ou sem valor de retorno.

É possível passar parâmetros, que deverão ser literais inteiros, a um programa em Ja através da linha de comandos. Supondo que o parâmetro formal do método `main()` é `args`, os respetivos 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 lógicos, inteiros, reais ou string.

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

O significado de um programa em Ja será o mesmo que o seu significado em Java 8. 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);  
    }  
}
```

¹ James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley, *The Java® Language Specification, Java SE 8 Edition*, Oracle America, Inc., 2015-02-13. [<https://docs.oracle.com/javase/specs/jls/se8/html/index.html>]

Metas

O projeto será estruturado como uma sequência de quatro metas com ponderação e datas de entrega próprias, a saber:

1. Análise lexical (10%) – até 1 de março de 2017
2. Análise sintática (30%) – até 31 de março de 2017
3. Análise semântica (30%) – até 5 de maio de 2017
4. Geração de código intermédio (30%) – até 2 de junho de 2017

Em cada uma das metas, o trabalho será obrigatoriamente validado no mooshak usando um concurso criado especificamente para o efeito. O código submetido ao mooshak deve estar devidamente comentado de modo a permitir compreender a estratégia de implementação adotada. Para além disso, a entrega final do projeto deverá ser feita no inforestudante até às **23h59** do dia **2 de junho**, e incluir todo o código fonte produzido no âmbito do projeto (exatamente os mesmos zip que tiverem sido submetidos atempadamente ao mooshak em cada meta, bem como os que, eventualmente, tiverem sido submetidos às pós-metas) e um ficheiro grupo.txt contendo os dados do grupo, no formato:

```
Grupo: <nome_do_grupo>
Nome1: <nome_aluno_1>
Numero1: <numero_aluno_1>
Email1: <email_aluno_1>
Nome2: <nome_aluno_2>
Numero2: <numero_aluno_2>
Email2: <email_aluno_2>
```

Os ficheiros zip correspondentes a cada submissão devem chamar-se 1.zip, ..., 4.zip para as submissões regulares, e 1pos.zip, 2pos.zip, ..., para as submissões às pós-metas.

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 será realizada **em grupo** e terá lugar entre os dias **5 e 16 de junho de 2017**. A nota final do projeto diz respeito à prestação **do grupo** na defesa e está limitada pela soma das pontuações obtidas no mooshak em cada uma das metas. A pontuação de cada meta será a obtida à respetiva data de entrega ou, quando esta for pelo menos 80% do total, a obtida na pós-meta à data de entrega final do projeto. *Excecionalmente*, e apenas por motivos justificados, a pontuação das pós-metas poderá ser tida em conta noutras situações, mas a classificação final nunca poderá exceder a pontuação obtida no mooshak para as diversas metas à data da última entrega.

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

Meta 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 Ja

ID : Sequências alfanuméricas começadas por uma letra, onde os símbolos “_” e “\$” contam como letras. Letras maiúsculas e minúsculas são consideradas letras diferentes.

DECLIT : o dígito zero, ou sequências de dígitos decimais ou “_”, começadas por um dígito diferente de zero e terminadas num dígito.

REALLIT : uma parte inteira seguida de um ponto, opcionalmente seguido de uma parte fracionária e/ou de um expoente; ou um ponto seguido de uma parte fracionária, opcionalmente seguida de um expoente; ou uma parte inteira seguida de um expoente. O expoente consiste numa das letras “e” ou “E” seguida de um número opcionalmente precedido de um dos sinais “+” ou “-”. Tanto a parte inteira como a parte fracionária e o número do expoente consistem em sequências de dígitos decimais ou “_” começadas e terminadas por um dígito.

STRLIT : uma sequência de caracteres (excepto “*carriage return*”, “*newline*”, ou aspas duplas) e/ou “sequências de escape” entre aspas duplas. Apenas as sequências de escape \f, \n, \r, \t, \\ e \" são definidas pela linguagem. Sequências de escape não definidas devem dar origem a erros lexicais, como se detalha mais adiante.

BOOL = “boolean”

BOOLLIT = “true” | “false”

CLASS = “class”

DO = “do”

DOTLENGTH = “.length”

DOUBLE = “double”

ELSE = “else”

IF = “if”

INT = “int”

PARSEINT = “Integer.parseInt”

PRINT = “System.out.println”

PUBLIC = “public”

RETURN = “return”

STATIC = “static”

STRING = “String”

VOID = “void”

WHILE = “while”

OCURV = “(”

CCURV = “)”

OBRACE = “{”

CBRACE = “}”

OSQUARE = “[”

CSQUARE = “]”

AND = “&&”

OR = “||”

LT = "<"
GT = ">"
EQ = "=="
NEQ = "!="
LEQ = "<="
GEQ = ">="
PLUS = "+"
MINUS = "-"
STAR = "*"
DIV = "/"
MOD = "%"
NOT = "!"
ASSIGN = "="
SEMI = ";"
COMMA = ","
RESERVED = keywords do Java 8 não utilizadas em Ja, bem como os operadores "++" e "--", o literal "null" e os identificadores "Integer" e "System".

Implementação

O analisador deverá chamar-se jac, ler o ficheiro a processar através do stdin e, se invocado com a opção "-l", emitir o resultado da análise para o stdout e terminar. Na ausência de qualquer opção, ou se invocado com a opção "-1", nada deve ser escrito no stdout à excepção de possíveis mensagens de erro². Caso o ficheiro Echo.ja contenha o programa de exemplo dado anteriormente, a invocação

```
./jac -l < Echo.ja
```

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)  
SEMI  
ID(x)  
ASSIGN  
PARSEINT  
OCURV
```

² O novo significado desta opção é válido apenas a partir do fim da Meta 1, e destina-se a garantir compatibilidade das soluções das Metas seguintes com a Pós-Meta 1.

```
ID(args)
OSQUARE
DECLIT(0)
CSQUARE
CCURV
SEMI
PRINT
OCURV
ID(x)
CCURV
SEMI
CBRACE
CBRACE
```

O analisador deve aceitar (e ignorar) espaço em branco (espaços, tabs e os caracteres “*form feed*”, “*carriage return*” e “*newline*”) bem como comentários dos tipos `/* . . . */` e `// . . .`. Deve ainda 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 DECLIT.

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 (<s>)\n"
"Line <num linha>, col <num coluna>: invalid escape sequence (<s>)\n"
"Line <num linha>, col <num coluna>: unterminated comment\n"
"Line <num linha>, col <num coluna>: unterminated string literal\n"
```

onde `<num linha>` e `<num coluna>` devem ser substituídos pelos valores correspondentes ao *início* do token que originou o erro, e `<s>` deve ser substituído por esse token. O analisador deve recuperar da ocorrência de erros lexicais a partir do fim desse token. No caso de uma string não terminada que inclua sequências de escape inválidas, o erro de string não terminada deve ser apresentado após os erros de sequência inválida.

Submissão

O trabalho deverá ser validado no mooshak, usando o concurso criado especificamente para o efeito em <https://mooshak2.dei.uc.pt/~comp2017>. 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 `jac.l` e ser colocado num ficheiro zip com o nome `jac.zip`. O ficheiro zip não deve conter quaisquer diretórios.

Meta II – Analisador sintático

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

Gramática inicial em notação EBNF

```
Program → CLASS ID OBRACE { FieldDecl | MethodDecl | SEMI } CBRACE
FieldDecl → PUBLIC STATIC Type ID { COMMA ID } SEMI
MethodDecl → PUBLIC STATIC MethodHeader MethodBody
MethodHeader → ( Type | VOID ) ID OCURV [ FormalParams ] CCURV
MethodBody → OBRACE { VarDecl | Statement } CBRACE
FormalParams → Type ID { COMMA Type ID }
FormalParams → STRING OSQUARE CSQUARE ID
VarDecl → Type ID { COMMA ID } SEMI
Type → BOOL | INT | DOUBLE
Statement → OBRACE { Statement } CBRACE
Statement → IF OCURV Expr CCURV Statement [ ELSE Statement ]
Statement → WHILE OCURV Expr CCURV Statement
Statement → DO Statement WHILE OCURV Expr CCURV SEMI
Statement → PRINT OCURV ( Expr | STRLIT ) CCURV SEMI
Statement → [ ( Assignment | MethodInvocation | ParseArgs ) ] SEMI
Statement → RETURN [ Expr ] SEMI
Assignment → ID ASSIGN Expr
MethodInvocation → ID OCURV [ Expr { COMMA Expr } ] CCURV
ParseArgs → PARSEINT OCURV ID OSQUARE Expr CSQUARE CCURV
Expr → Assignment | MethodInvocation | ParseArgs
Expr → Expr ( AND | OR ) Expr
Expr → Expr ( EQ | GEQ | GT | LEQ | LT | NEQ ) Expr
Expr → Expr ( PLUS | MINUS | STAR | DIV | MOD ) Expr
Expr → ( PLUS | MINUS | NOT ) Expr
Expr → ID [ DOTLENGTH ]
Expr → OCURV Expr CCURV
Expr → BOOLLIT | DECLIT | REALLIT
```

Uma vez que a gramática dada é ambígua e é apresentada em notação EBNF, onde [...] representa “opcional” e {...} representa “**sequência de zero ou mais**,” 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 Ja e Java.

Implementação

O analisador deverá chamar-se `jac`, ler o ficheiro a processar através do `stdin` e emitir quaisquer resultados para o `stdout`. Por uma questão de compatibilidade com a meta anterior, se o analisador for invocado com uma das opções `-l` ou `-1`, deverá realizar apenas a análise lexical, emitir o resultado dessa análise para o `stdout` (erros lexicais e, no caso da opção `-l`, os tokens encontrados) e terminar.

Se não for passada qualquer **dessas opções**, o analisador deve detetar a existência de quaisquer erros lexicais ou de sintaxe no ficheiro de entrada, e emitir as mensagens de erro correspondentes para o stdout.

Tratamento e recuperação 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 sejam encontrados erros de sintaxe, o analisador deve imprimir mensagens de erro com o seguinte formato:

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

A analisador deve ainda implementar recuperação local de erros de sintaxe através da adição das seguintes regras de erro à gramática (ou de outras com o mesmo efeito dependendo das alterações que a gramática dada vier a sofrer):

```
FieldDecl → error SEMI  
Statement → error SEMI  
ParseArgs → PARSEINT OCURV error CCURV  
MethodInvocation → ID OCURV error CCURV  
Expr → OCURV error CCURV
```

Árvore de sintaxe abstrata (AST)

Caso o ficheiro gcd.ja contenha o programa:

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

a invocação

```
./jac -t < gcd.ja
```

deverá gerar a árvore de sintaxe abstrata correspondente, e imprimi-la no `stdout` conforme a seguir se explica. 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.

As árvores de sintaxe abstrata geradas durante a análise sintática devem incluir apenas nós dos tipos indicados 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 { FieldDecl | MethodDecl } )
```

Declaração de variáveis

```
FieldDecl(2)          ( <type> Id )  
VarDecl(2)            ( <type> Id )
```

Definição de métodos

```
MethodDecl(2)          ( MethodHeader MethodBody )  
MethodHeader(3)         ( ( <type> | Void ) Id MethodParams )  
MethodParams(>=0)       ( { ParamDecl } )  
ParamDecl(2)            ( ( <type> | StringArray ) Id )  
MethodBody(>=0)         ( { VarDecl | <statement> } )
```

Statements

```
Block(!=1) DoWhile(2) If(3) Print(1) Return(<=1) While(2) Assign(2)  
Call(>=1) ParseArgs(2)
```

Operadores

```
Assign(2) 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) Call(>=1) ParseArgs(2)
```

Terminais

```
Bool BoolLit Double Declit Id Int RealLit StrLit Void
```

Nota: não deverão ser gerados nós supérfluos, nomeadamente `Block` com menos de dois statements no seu interior, exceto para representar um statement obrigatório que seja vazio. Os nós `Program`, `MethodParams` e `MethodBody`, não deverão ser considerados redundantes independentemente do número de nós filhos.

No caso do programa dado, o resultado deve ser:

```
Program  
..Id(gcd)  
..MethodDecl  
...MethodHeader  
.....Void  
.....Id(main)  
.....MethodParams  
.....ParamDecl  
.....StringArray  
.....Id(args)  
...MethodBody  
.....VarDecl  
.....Int  
.....Id(a)
```



```

.....VarDecl
.....Int
.....Id(b)
.....Assign
.....Id(a)
.....ParseArgs
.....Id(args)
.....DeclLit(0)
.....Assign
.....Id(b)
.....ParseArgs
.....Id(args)
.....DeclLit(1)
.....If
.....Eq
.....Id(a)
.....DeclLit(0)
.....Print
.....Id(b)
.....Block
.....While
.....Gt
.....Id(b)
.....DeclLit(0)
.....If
.....Gt
.....Id(a)
.....Id(b)
.....Assign
.....Id(a)
.....Sub
.....Id(a)
.....Id(b)
.....Assign
.....Id(b)
.....Sub
.....Id(b)
.....Id(a)
.....Print
.....Id(a)

```

Desenvolvimento do analisador

Sugere-se que o desenvolvimento do analisador seja efetuado em duas fases. A primeira deverá visar a tradução da gramática para o yacc de modo a permitir detetar e reportar eventuais erros de sintaxe. A segunda deverá incidir sobre a construção da árvore de sintaxe abstrata e sua impressão na saída.

Toda a memória alocada durante a execução do analisador deve ser libertada antes deste terminar, devendo ser dada especial atenção aos casos em que a construção da AST é interrompida devido à ocorrência de erros de sintaxe. Aconselha-se vivamente a utilização do analisador estático do Clang (scan-build) e do *memory debugger/profiler* valgrind durante o processo de desenvolvimento.

Submissão

O trabalho deverá ser validado no mooshak, usando o concurso criado especificamente para o efeito em <https://mooshak2.dei.uc.pt/~comp2017>. 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 `jac.l` e `jac.y` e ser colocados juntamente com quaisquer ficheiros adicionais necessários à compilação do analisador num único ficheiro zip com o nome `jac.zip`. O ficheiro zip não deve conter quaisquer diretórios.