

Projeto de Compiladores 2019/20

Compilador para a linguagem Juc

20 de abril de 2020

Este projeto consiste no desenvolvimento de um compilador para a linguagem Juc, que é um subconjunto da linguagem Java de acordo com a especificação Java SE 9 (disponível na página <https://docs.oracle.com/javase/specs/jls/se9/html/index.html>).

Na linguagem Juc é possível usar variáveis e literais dos tipos `boolean`, `int` e `double` (estes dois últimos com sinal). É também possível usar literais do tipo `String`, apenas para efeito de escrita no *stdout*. A linguagem Juc inclui expressões aritméticas e lógicas, instruções de atribuição, operadores relacionais e instruções de controlo (`while` e `if-else`). Inclui também métodos estáticos com os tipos de dados já referidos e ainda o tipo especial `String[]`, sendo a passagem de parâmetros sempre feita por valor e podendo ou não ter valor de retorno. A ausência de valor de retorno é identificada pela palavra-chave `void`.

Os programas da linguagem Juc são compostos por uma única classe (principal) contendo métodos e atributos, todos eles estáticos. O método `main(...)` invocado no início de cada programa pode receber parâmetros, que deverão ser literais inteiros, através da linha de comandos. Supondo que o parâmetro formal do método `main(...)` é `args`, os respetivos valores podem ser obtidos através do método pré-definido `Integer.parseInt(args[...])` e a expressão `args.length` dá o número de parâmetros. O método pré-definido `System.out.print(...)` permite escrever na consola valores lógicos, inteiros, reais e strings.

O significado de um programa na linguagem Juc será o mesmo que na linguagem Java, assumindo a pré-definição dos métodos `Integer.parseInt(...)` e `System.out.print(...)`, bem como da construção `.length`. Por fim, são aceites comentários nas formas `/* ... */` e `// ...` que deverão ser ignorados. Assim, por exemplo, o programa que se segue calcula o fatorial de um número passado como argumento:

```
class Factorial {
    public static int factorial(int n) {
        if (n == 0)
            return 1;
        return n * factorial(n-1);
    }

    public static void main(String[] args) {
        int argument;
        argument = Integer.parseInt(args[0]);
        System.out.print(factorial(argument));
    }
}
```

O programa anterior declara uma variável `argument` do tipo `int` e atribui-lhe o valor inteiro do argumento passado ao programa, usando o método `parseInt(...)` para realizar a conversão. De seguida, calcula o fatorial desse valor e invoca o método `print(...)` para escrever o resultado na consola.

1 Metas e avaliação

O projeto está estruturado em quatro metas encadeadas, nas quais o resultado de cada meta é o ponto de partida para a meta seguinte. As datas e as ponderações são as seguintes:

1. Análise lexical (19%) – 11 de março de 2020
2. Análise sintática (25%) – 3 de abril de 2020 (meta de avaliação)
3. Análise semântica (25%) – 8 de maio de 2020
4. Geração de código (25%) – 29 de maio de 2020 (meta de avaliação)

A entrega final será acompanhada de um relatório que tem um peso de 6% na avaliação. Para além disso, a entrega final do trabalho deverá ser feita através do Inforestudante, até ao dia seguinte ao da Meta 4, e incluir todo o código-fonte produzido no âmbito do projeto (exatamente os mesmos arquivos .zip que tiverem sido colocados no MOOSHAK em cada meta).

O trabalho será verificado no MOOSHAK em cada uma das metas usando um concurso criado para o efeito. A classificação final da Meta 1 é obtida em conjunto com a Meta 2 e a classificação final da Meta 3 é obtida em conjunto com a Meta 4. O nome do grupo a registar no MOOSHAK é obrigatoriamente da forma “uc2019123456_uc2019654321” usando os números de estudante como identificação do grupo na página <http://mooshak2.dei.uc.pt/~comp2020> na qual o MOOSHAK está acessível. Será tida em conta apenas a última submissão ao problema A de cada concurso do MOOSHAK para efeitos de avaliação.

1.1 Defesa e grupos

O trabalho será realizado por grupos de dois alunos inscritos em turmas práticas do mesmo docente. Em casos excecionais, a confirmar com o docente, admite-se trabalhos individuais. A defesa oral do trabalho será realizada em grupo na semana seguinte à entrega da Meta 4. A nota final do projeto é limitada pela soma ponderada das pontuações obtidas no MOOSHAK em cada uma das metas e diz respeito à prestação individual na defesa. Assim, a classificação final nunca poderá exceder a pontuação obtida no MOOSHAK acrescida da classificação do relatório final. Aplica-se mínimos de 40% à nota final após a defesa. Os programas de teste colocados no repositório <https://git.dei.uc.pt/rbarbosa/Comp2020/tree/master> por cada estudante serão contabilizados na avaliação.

2 Meta 1 – Analisador lexical

Nesta primeira meta deve ser programado um analisador lexical para a linguagem Juc. A programação deve ser feita em linguagem C utilizando a ferramenta *lex*. Os “tokens” a ser considerados são apresentados de seguida e deverão estar de acordo com a especificação da linguagem Java, disponível em <https://docs.oracle.com/javase/specs/jls/se9/html/jls-3.html> na sua versão original.

2.1 Tokens da linguagem Juc

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

INTLIT: representa uma constante inteira composta pelo dígito zero, ou seqüências 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 seqüências de dígitos decimais ou “_” começadas e terminadas por um dígito.

STRLIT: uma seqüência de caracteres (exceto “carriage return”, “newline” e aspas duplas) e/ou “seqüências de escape” entre aspas duplas. Apenas as seqüências de escape \f, \n, \r, \t, \\ e \" são especificadas pela linguagem. Seqüências de escape não especificadas devem dar origem a erros lexicais, tal como se detalha mais adiante.

BOOLLIT = “true” | “false”

AND = “&&”

ASSIGN = “=”

STAR = “*”

COMMA = “,”

DIV = “/”

EQ = “==”

GE = “>=”

GT = “>”

LBRACE = “{”

LE = “<=”

LPAR = “(”

LSQ = “[”

LT = “<”

MINUS = “-”

MOD = “%”

NE = “!=”

NOT = “!”

OR = “||”

PLUS = “+”

RBRACE = “}”

RPAR = “)”

RSQ = “]”

SEMICOLON = “;”

ARROW = “->”

LSHIFT = “<<”

RSHIFT = “>>”

XOR = “^”

BOOL = “boolean”

CLASS = “class”

DOTLENGTH = “.length”

DOUBLE = “double”

ELSE = “else”

IF = “if”

INT = “int”

PRINT = “System.out.print”

PARSEINT = “Integer.parseInt”

PUBLIC = “public”

RETURN = “return”

STATIC = “static”

STRING = “String”

VOID = “void”

WHILE = “while”

RESERVED: palavras reservadas da linguagem Java não utilizadas em Juc bem como o operador de incremento (“++”), o operador de decremento (“--”), o literal “null” e os identificadores “Integer” e “System”.

2.2 Programação do analisador

O analisador deverá chamar-se *jucompiler*, ler o ficheiro a processar através do *stdin* e, quando invocado com a opção *-l*, deve emitir os tokens e as mensagens de erro para o *stdout* e terminar. Na ausência de qualquer opção, ou se invocado com a opção *-e1*, deve escrever no *stdout* apenas as mensagens de erro. Por exemplo, caso o ficheiro *Factorial.java* contenha o programa de exemplo dado anteriormente, que calcula o fatorial de números, a invocação:

```
jucompiler -l < Factorial.java
```

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

```
CLASS
ID(Factorial)
LBRACE
PUBLIC
STATIC
INT
ID(factorial)
LPAR
INT
ID(n)
RPAR
LBRACE
...
```

Figura 1: Exemplo de output do analisador lexical. O output completo está disponível em: <https://git.dei.uc.pt/rbarbosa/Comp2020/blob/master/meta1/Factorial.out>

O analisador deve aceitar (e ignorar) como separador de tokens o espaço em branco (espaços, tabs e mudanças de linha), 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 exemplificado na figura acima para *ID*.

2.3 Tratamento de erros

Caso o ficheiro contenha erros lexicais, o programa deverá imprimir exatamente uma das seguintes mensagens no *stdout*, consoante o caso:

```
Line <num linha>, col <num coluna>: illegal character (<c>)\n
Line <num linha>, col <num coluna>: invalid escape sequence (<c>)\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 <c> devem ser substituídos por esse token. Tanto as linhas como as colunas são numeradas a partir de 1. O analisador deve recuperar da ocorrência de erros lexicais a partir do *fim* do respetivo 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.

2.4 Entrega da Meta 1

O ficheiro *lex* a entregar deverá obrigatoriamente identificar os autores num comentário no topo desse ficheiro, contendo o nome e o número de estudante de cada elemento do grupo. Esse ficheiro deverá chamar-se *jucompiler.l* e ser enviado num arquivo de nome *jucompiler.zip* que não deverá ter quaisquer diretorias.

O trabalho deverá ser verificado no MOOSHAK usando o concurso criado especificamente para o efeito. Será tida em conta apenas a última submissão ao problema A desse concurso. Os restantes problemas destinam-se a ajudar na verificação do analisador. No entanto, o MOOSHAK não deve ser utilizado como ferramenta de depuração. Os estudantes devem usar e contribuir para o repositório disponível em <https://git.dei.uc.pt/rbarbosa/Comp2020/tree/master> contendo casos de teste. A página do MOOSHAK está indicada na Secção 1.

3 Meta 2 – Analisador sintático

O analisador sintático deve ser programado em C utilizando as ferramentas `lex` e `yacc`. A gramática que se segue especifica a sintaxe da linguagem Juc.

3.1 Gramática inicial em notação EBNF

```
Program → CLASS ID LBRACE { MethodDecl | FieldDecl | SEMICOLON } RBRACE
MethodDecl → PUBLIC STATIC MethodHeader MethodBody
FieldDecl → PUBLIC STATIC Type ID { COMMA ID } SEMICOLON
Type → BOOL | INT | DOUBLE
MethodHeader → ( Type | VOID ) ID LPAR [ FormalParams ] RPAR
FormalParams → Type ID { COMMA Type ID }
FormalParams → STRING LSQ RSQ ID
MethodBody → LBRACE { Statement | VarDecl } RBRACE
VarDecl → Type ID { COMMA ID } SEMICOLON
Statement → LBRACE { Statement } RBRACE
Statement → IF LPAR Expr RPAR Statement [ ELSE Statement ]
Statement → WHILE LPAR Expr RPAR Statement
Statement → RETURN [ Expr ] SEMICOLON
Statement → [ ( MethodInvocation | Assignment | ParseArgs ) ] SEMICOLON
Statement → PRINT LPAR ( Expr | STRLIT ) RPAR SEMICOLON
MethodInvocation → ID LPAR [ Expr { COMMA Expr } ] RPAR
Assignment → ID ASSIGN Expr
ParseArgs → PARSEINT LPAR ID LSQ Expr RSQ RPAR
Expr → Expr ( PLUS | MINUS | STAR | DIV | MOD ) Expr
Expr → Expr ( AND | OR | XOR | LSHIFT | RSHIFT ) Expr
Expr → Expr ( EQ | GE | GT | LE | LT | NE ) Expr
Expr → ( MINUS | NOT | PLUS ) Expr
Expr → LPAR Expr RPAR
Expr → MethodInvocation | Assignment | ParseArgs
Expr → ID [ DOTLENGTH ]
Expr → INTLIT | REALLIT | BOOLLIT
```

A gramática apresentada é ambígua e está escrita em notação EBNF, com a qual [...] significa “opcional” e {...} significa “zero ou mais repetições”. Portanto, a gramática deverá ser transfor-

mada para permitir a análise sintática ascendente com o yacc. Será necessário ter em conta as regras de associação dos operadores e as precedências, entre outros aspetos, de modo a garantir a compatibilidade entre as linguagens Juc e Java.

3.2 Programação do analisador

O analisador deverá chamar-se `jucompiler`, ler o ficheiro a processar através do *stdin* e emitir todos os resultados para o *stdout*. Quando invocado com a opção `-t` deve imprimir a árvore de sintaxe tal como se especifica nas secções que se seguem. Se invocado com a opção `-e2` deve escrever no *stdout* apenas as mensagens de erro relativas aos erros sintáticos e lexicais.

Para manter a compatibilidade com a fase anterior, se o analisador for invocado com uma das opções `-l` ou `-e1` deverá apenas realizar a análise lexical, emitir o resultado para o *stdout* (erros lexicais e no caso da opção `-l` também os tokens encontrados) e terminar. Se não for passada qualquer opção, o analisador deve apenas escrever no *stdout* as mensagens de erro correspondentes aos erros lexicais e de sintaxe.

3.3 Tratamento e recuperação de erros

Caso o ficheiro de entrada contenha erros lexicais, o programa deverá imprimir no *stdout* as mensagens já especificadas na meta 1 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, coluna e pelo valor semântico do *token* que dá origem ao erro. Isto pode ser conseguido escrevendo a função:

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

O analisador deve ainda incluir recuperação local de erros de sintaxe através da inclusão das seguintes regras de erro na gramática (ou de outras com o mesmo efeito, dependendo das alterações que a gramática dada venha a sofrer):

```
FieldDecl → error SEMICOLON  
Statement → error SEMICOLON  
ParseArgs → PARSEINT LPAR error RPAR  
MethodInvocation → ID LPAR error RPAR  
Expr → LPAR error RPAR
```

3.4 Árvore de sintaxe abstrata (AST)

Caso seja feita a seguinte invocação:

```
jucompiler -t < Factorial.java
```


deverá gerar a árvore de sintaxe abstrata correspondente e imprimi-la no stdout de acordo com a descrição que se segue. 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 abaixo indicados. 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) If(3) While(2) Return(<=1) Call(>=1) Print(1) ParseArgs(2) Assign(2)
```

Operadores

```
Assign(2) Or(2) And(2) Eq(2) Ne(2) Lt(2) Gt(2) Le(2) Ge(2) Add(2)
```

```
Sub(2) Mul(2) Div(2) Mod(2) Lshift(2) Rshift(2) Xor(2) Not(1) Minus(1)
```

```
Plus(1) Length(1) Call(>=1) ParseArgs(2)
```

Terminais

```
Bool BoolLit Double DeclLit Id Int RealLit StrLit StringArray 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.

A Figura 2 exemplifica a impressão da árvore de sintaxe abstrata do programa apresentado na primeira página, que calcula o fatorial de números.

```
Program
..Id(Factorial)
..MethodDecl
....MethodHeader
.....Int
.....Id(factorial)
.....MethodParams
.....ParamDecl
.....Int
.....Id(n)
....MethodBody
...
```

Figura 2: Exemplo de output do analisador sintático. O output completo está disponível em <https://git.dei.uc.pt/rbarbosa/Comp2020/blob/master/meta2/Factorial.out>

3.5 Desenvolvimento do analisador

Sugere-se que desenvolva o analisador de forma faseada. Deverá começar por re-escrever para o yacc a gramática acima apresentada de modo a detetar erros de sintaxe (isto é, inicialmente sem a árvore de sintaxe). Após terminada esta fase, e já garantindo que a gramática está correta, deverá focar-se no desenvolvimento do código necessário para a construção da árvore de sintaxe abstrata e a sua impressão para o stdout. O relatório final deverá descrever as opções tomadas na escrita da gramática, pelo que se recomenda agora a documentação dessa parte.

Para promover uma boa divisão de tarefas entre membros do grupo, sugere-se que comecem por analisar produções diferentes do topo da gramática. Outra possibilidade seria um membro começar pelo topo da gramática, em Program, e o outro membro começar pela base, em Expr. Teriam de coordenar o trabalho a partir do momento em que chegassem a não-terminais comuns na gramática.

Deverá ter em atenção que toda a memória alocada durante a execução do analisador deve ser libertada antes deste terminar, devendo ter em conta as situações em que a construção da AST é interrompida por erros de sintaxe.

3.6 Entrega da Meta 2

O ficheiro *lex* entregue deverá obrigatoriamente listar os autores num comentário colocado no topo desse ficheiro, contendo o nome e o número de estudante de cada membro do grupo. Os ficheiros *lex* e *yacc* a entregar deverão chamar-se *jucompiler.l* e *jucompiler.y* e ser colocados num único arquivo com o nome *jucompiler.zip* juntamente com quaisquer outros ficheiros necessários para compilar o analisador.

O trabalho deverá ser avaliado no MOOSHAK, usando o concurso criado especificamente para o efeito e cuja página está acima indicada na Secção 1. Para efeitos de avaliação, 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. No entanto, o MOOSHAK não deve ser utilizado como ferramenta de depuração. Os estudantes deverão usar e contribuir para o repositório disponível em <https://git.dei.uc.pt/rbarbosa/Comp2020/tree/master> contendo casos de teste.

4 Meta 3 – Analisador semântico

O analisador semântico deve ser programado em C tendo por base o analisador sintático desenvolvido na meta anterior com as ferramentas `lex` e `yacc`. O analisador deverá chamar-se `jucompiler`, ler o ficheiro a processar através do `stdin` e detetar a ocorrência de quaisquer erros (lexicais, sintáticos ou semânticos) no ficheiro de entrada. Considere a invocação

```
./jucompiler < Factorial.java
```

deverá levar o analisador a proceder à análise lexical e sintática do programa, e caso este seja válido, proceder à análise semântica.

Por uma questão de compatibilidade com a fase anterior, se o analisador for invocado com a opção `-t`, deverá realizar *apenas* a análise sintática, e emitir o resultado para o `stdout` (erros lexicais e/ou sintáticos e, no caso da opção `-t`, a árvore de sintaxe abstrata se não houver erros de sintaxe) e terminar *sem* proceder à análise semântica.

Sendo o programa sintaticamente válido, a invocação

```
./jucompiler -s < Factorial.java
```

deve fazer com que o analisador imprima no `stdout` a(s) tabela(s) de símbolos correspondentes seguida(s) de uma linha em branco e da árvore de sintaxe abstrata anotada com os tipos das variáveis, funções e expressões, como a seguir se especifica.

4.1 Tabelas de símbolos

Durante a análise semântica, deve ser construída uma tabela de símbolos global, bem como os identificadores das variáveis e/ou métodos definidos no programa. Além disso, devem ser construídas tabelas de símbolos correspondentes aos métodos, que deverão conter os identificadores dos respetivos parâmetros formais e variáveis locais.

Por sua vez, as tabelas correspondentes às funções definidas no programa irão conter a string “return” (usada para representar o valor de retorno) e os identificadores dos respetivos parâmetros formais e variáveis locais. Note que caso uma função não tenha tipo de retorno, deverá à mesma ser inserida na tabela com o tipo de dados `none`.

Para o programa de exemplo dado, as tabelas de símbolos a imprimir são as que se seguem. O formato das linhas é “Name\t[ParamTypes]\tType[\tparam]”, onde `[]` significa *opcional*.

```
==== Class Factorial Symbol Table ====
factorial      (int)      int
main          (String[])  void

==== Method factorial(int) Symbol Table ====
return         int
n              int        param

==== Method main(String[]) Symbol Table ====
return         void
args           String[]   param
argument      int
```

Os símbolos (e as tabelas) devem ser apresentados por ordem de declaração no programa fonte. No essencial, a notação para os tipos segue as convenções de Go. Deve ser deixada uma linha em branco entre tabelas consecutivas, e entre as tabelas e a árvore de sintaxe abstrata anotada.

4.2 Árvore de sintaxe anotada

Para o programa dado, a árvore de sintaxe abstrata anotada a imprimir a seguir às tabelas de símbolos quando é dada a opção `-s` seria a seguinte:

```
Program
..Id(Factorial)
..MethodDecl
....MethodHeader
.....Int
.....Id(factorial)
.....MethodParams
.....ParamDecl
.....Int
.....Id(n)
....MethodBody
.....If
.....Eq - boolean
.....Id(n) - int
.....Declit(0) - int
.....Return
.....Declit(1) - int
...
```

Figura 3: Exemplo de output do analisador semântico. O output completo está disponível em <https://git.dei.uc.pt/rbarbosa/Comp2020/blob/master/meta3/Factorial.out>

Deverão ser anotados apenas os nós correspondentes a expressões. Declarações ou statements que não sejam expressões não devem ser anotados.

4.3 Tratamento de erros semânticos

Eventuais erros de semântica deverão ser detetados e reportados no stdout de acordo com o catálogo de erros abaixo listados, onde cada mensagem deve ser antecidida pelo prefixo “Line <linha>, col <coluna>: ” e terminada com um caractere de fim de linha.

```
Symbol <token> already defined
Symbol _ is reserved
Cannot find symbol <token>
Operator <token> cannot be applied to type <type>
Operator <token> cannot be applied to types <type>, <type>
Incompatible type <type> in <token> statement
Number <token> out of bounds
Reference to method <token> is ambiguous
```

Caso seja detetado algum erro durante a análise semântica do programa, **o analisador deverá imprimir a mensagem de erro apropriada e continuar, dando o pseudo-tipo** `undef` a quaisquer símbolos desconhecidos e aos resultados de operações cujo tipo não possa ser determinado devido aos seus operandos (inválidos), o que pode dar origem a novos erros semânticos. Os tipos de dados (`<type>`) a reportar nas mensagens de erro deverão ser os mesmos usados na impressão das tabelas de símbolos, e todos os tokens (`<token>`) deverão ser apresentados tal como aparecem no código fonte. Os números de linha e coluna a reportar dizem respeito ao primeiro carácter dos seguintes tokens:

- o identificador que dá origem ao erro,
- o operador cujos argumentos são de tipos incompatíveis,
- o operador ou o identificador da função invocada correspondente à raiz da AST da expressão que é incompatível com a forma como é usada num statement,
- o identificador da função invocada quando o número de parâmetros estiver errado,
- o operando ou constante que dá origem ao erro,
- o token `return`, apenas quando este não for seguido de uma expressão e for esperado um valor de retorno, sendo que neste caso a mensagem de erro a imprimir será “Incompatible type void in return statement”, apesar de se tratar de um abuso de linguagem.

A impressão das tabelas de símbolos e da AST anotada (se for o caso) deve ser feita depois da impressão de todas as mensagens de erro. Adicionalmente, faz-se notar:

- `Integer.parseInt` e `.length` devem ser entendidos como operadores cujo resultado é do tipo `int`, e `System.out.println` deve ser entendido como um statement.
- Não é possível realizar qualquer operação sobre objetos do tipo `String[]`, à exceção de `Integer.parseInt`, `.length`, e passagem a um método com um parâmetro formal de igual tipo.
- Apesar de em Java ser permitido escrever “-2147483648” (mas não “-(2147483648)”!), o literal `2147483648` deverá sempre originar o erro “Number 2147483648 out of bounds”, uma vez que, devido à simplificação das expressões entre parêntesis na construção da AST, não é possível distinguir entre os dois casos referidos.
- A seleção do método invocado deve seguir as seguintes regras (simplificadas):
 - Se existir um método com o mesmo número e tipo de parâmetros formais que os dos parâmetros reais passados na invocação, é invocado esse método.
 - Caso contrário:
 - * Se existir um único método com número de parâmetros formais igual ao de parâmetros reais passados na invocação, e os tipos dos parâmetros reais são compatíveis com os dos parâmetros formais correspondentes, é selecionado esse método.
 - * Se existir mais do que um método com as características descritas no ponto anterior, é gerado o erro “Reference to method <symbol> is ambiguous”. Caso contrário, é gerado o erro “Cannot find symbol <symbol>”.
- Quando não for possível determinar qual o método invocado, ou por não existir método compatível, ou por a invocação ser ambígua, os nós `Call` e `Id` correspondentes devem ser anotados com `undef`.

4.4 Programação do analisador

Sugere-se que o desenvolvimento do analisador seja efetuado em três fases. A primeira deverá consistir na construção das tabelas de símbolos e sua impressão, a segunda na verificação de tipos e anotação da AST e a terceira no tratamento de erros semânticos.

4.5 Entrega da Meta 3

O trabalho deverá ser avaliado no MOOSHAK, usando o concurso criado especificamente para o efeito e cuja página está acima indicada na Secção 1. Para efeitos de avaliação, 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 semântica e à construção da árvore de sintaxe abstrata anotada, de acordo com a estratégia de desenvolvimento proposta. No entanto, o MOOSHAK não deve ser utilizado como ferramenta de depuração. Os estudantes deverão usar e contribuir para o repositório disponível em <https://git.dei.uc.pt/rbarbosa/Comp2020/tree/master> contendo casos de teste.

Os ficheiros `lex` e `yacc` a apresentar deverão chamar-se `jucompiler.l` e `jucompiler.y` e ser colocados juntamente com quaisquer ficheiros adicionais necessários à compilação do analisador num único ficheiro `.zip` com o nome `jucompiler.zip`. O ficheiro `.zip` não deve conter quaisquer diretorias. Note que deverá *listar os autores em comentário* no ficheiro `jucompiler.l`.

5 Entrega final e relatório

A entrega final do projeto será feita no Inforestudante até ao dia seguinte ao da Meta 4, e deve incluir todo o código-fonte produzido no âmbito do projeto: precisamente os quatro arquivos `.zip` que tiverem sido apresentados no MOOSHAK em cada meta. Os ficheiros `.zip` correspondentes a cada submissão devem chamar-se `1.zip`, `2.zip`, `3.zip`, `4.zip`, para as submissões às Metas 1, 2, 3 e 4, respetivamente.

Em todas as entregas no MOOSHAK o ficheiro `gocompiler.l` deve identificar os autores num comentário acrescentado ao topo do ficheiro. Sem a identificação dos autores de cada trabalho não será possível atribuir a respetiva classificação.

O relatório final terá três secções limitadas a 1200 palavras (400 palavras por cada secção), sendo que deverá documentar concisamente as opções técnicas relativas

- (i) à gramática re-escrita,
- (ii) aos algoritmos e estruturas de dados da AST e da tabela de símbolos, e
- (iii) à geração de código.