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

v4.0.1

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 erro[2]. 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 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.

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)
..........DecLit(0)
......Assign
........Id(b)
........ParseArgs
..........Id(args)
..........DecLit(1)
......If
........Eq
..........Id(a)
..........DecLit(0)
........Print
..........Id(b)
........Block
..........While
............Gt
..............Id(b)
..............DecLit(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.

Fase III – Análise semântica

A análise semântica da linguagem Ja deve ser implementada em C tendo por base o analisador sintático desenvolvido com as ferramentas `lex` e `yacc` na fase anterior. O analisador deverá chamar-se `jac`, ler o ficheiro a processar através do `stdin`, e detetar a existência de quaisquer erros (lexicais, de sintaxe, ou de semântica) no ficheiro de entrada. Caso o ficheiro `gcd2.ja` contenha o programa:

```
class gcd2 {
  public static int gcd;
  public static void main(String[] args) {
    int a, b;
    if (args.length >= 2) {
      a = Integer.parseInt(args[0]);
      b = Integer.parseInt(args[1]);
      gcd = gcd(a, b);
      System.out.println(gcd);
    } else
      System.out.println("Error: two parameters required.");
  }
  public static int gcd(int a, int b) {
    if (a == 0)
        return b;
    else {
        while (b > 0)
            if (a > b)
                a = a-b;
            else
                b = b-a;
        return a;
    }
  }
}
```

a invocação

```
./jac < gcd2.ja
```

deverá levar o analisador a proceder à análise sintática do programa e, sendo este sintaticamente válido, a proceder também à análise semântica. Erros lexicais deverão ser tolerados (e reportados como até aqui) desde que não originem também erros de sintaxe.

Por uma questão de compatibilidade com a fase anterior, se o analisador for invocado com uma das opções `-t` ou `-2`, deverá realizar apenas a análise sintática (e a análise lexical subjacente), 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 a análise semântica.

Sendo o programa sintaticamente válido, a invocação

```
./jac -s < gcd2.ja
```

deve levar o analisador a imprimir 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, etc, como a seguir se especifica.

Tabelas de símbolos

Durante a análise semântica, deve ser construída uma tabela de símbolos principal contendo os identificadores dos atributos e/ou métodos da classe definidos no programa fonte. Além disso, devem ser construídas tabelas de símbolos correspondentes aos métodos, que deverão conter os identificadores dos respetivos parâmetros formais e variáveis locais.

Para o programa de exemplo dado, as tabelas de símbolos a imprimir são as seguintes. O formato das linhas é "`Name\t[ParamTypes]\tType[\tFlag]`", onde `[]` quer dizer opcional.

```
===== Class gcd2 Symbol Table =====
gcd         int
main  (String[])  void
gcd   (int,int)   int

===== Method main(String[]) Symbol Table =====
return            void
args        String[]    param
a           int
b           int

===== Method gcd(int,int) Symbol Table =====
return            int
a           int   param
b           int   param
```

Os símbolos (e as tabelas) devem ser apresentados por ordem de primeira declaração no programa fonte. A string "`return`" é usada para representar o valor de retorno dos métodos e a string "`String[]`" é usada para indicar o tipo do identificador (argumento do método `main`, ou de outro método invocado a partir deste) que permite aceder aos parâmetros passados na linha de comandos. Deve ser deixada uma linha em branco entre tabelas consecutivas, e entre as tabelas e a árvore de sintaxe abstrata anotada.

Árvore de sintaxe abstrata 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(gcd2)
..FieldDecl
....Int
....Id(gcd)
..MethodDecl
....MethodHeader
......Void
......Id(main)
......MethodParams
........ParamDecl
..........StringArray
..........Id(args)
```

```
....MethodBody
......VarDecl
........Int
........Id(a)
......VarDecl
........Int
........Id(b)
......If
........Geq - boolean
..........Length - int
............Id(args) - String[]
..........DecLit(2) - int
........Block
..........Assign - int
............Id(a) - int
............ParseArgs - int
..............Id(args) - String[]
..............DecLit(0) - int
..........Assign - int
............Id(b) - int
............ParseArgs - int
..............Id(args) - String[]
..............DecLit(1) - int
..........Assign - int
............Id(gcd) - int
............Call - int
..............Id(gcd) - (int,int)
..............Id(a) - int
..............Id(b) - int
..........Print
............Id(gcd) - int
........Print
..........StrLit("Error: two parameters required.") - String
..MethodDecl
....MethodHeader
......Int
......Id(gcd)
......MethodParams
........ParamDecl
..........Int
..........Id(a)
........ParamDecl
..........Int
..........Id(b)
....MethodBody
......If
........Eq - boolean
..........Id(a) - int
..........DecLit(0) - int
........Return
..........Id(b) - int
........Block
..........While
............Gt - boolean
..............Id(b) - int
..............DecLit(0) - int
............If
..............Gt - boolean
................Id(a) - int
................Id(b) - int
..............Assign - int
................Id(a) - int
................Sub - int
..................Id(a) - int
```

```
.................Id(b) - int
.............Assign - int
.............Id(b) - int
.............Sub - int
.................Id(b) - int
.................Id(a) - int
.........Return
...........Id(a) - int
```

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

Tratamento de erros semânticos

Eventuais erros de semântica deverão ser detetados e reportados no stdout de acordo com o catálogo de erros abaixo, onde cada mensagem deve ser antecedida pelo prefixo "`Line <linha>, col <coluna>: `" e terminada com um caracter de fim de linha.

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

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 tipos (incompatíveis) dos seus operandos, o que pode dar origem a novos erros semânticos. De resto, os tipos de dados (`<type>`) a reportar nas mensagens de erro deverão ser os mesmos usados na impressão das tabelas de símbolos e da AST anotada, e todos os tokens (`<token>`) deverão ser apresentados tal como aparecem no código fonte. Os símbolos (`<symbol>`) a reportar deverão consistir, ou no nome de um atributo ou variável, ou no nome de um método com a especificação dos tipos dos respetivos parâmetros formais. Por exemplo, a invocação "`f(0)`" poderia originar o erro:

```
Line 10, col 25: Cannot find symbol f(int)
```

Os números de linha e coluna a reportar dizem respeito ao primeiro caracter dos seguintes tokens:

- O identificador que dá origem ao erro
- O operador cujos argumentos são de tipos incompatíveis
- O operador ou o identificador do método invocado correspondente à raiz da AST da expressão que é incompatível com a forma como é usada num statement
- O token `return`, apenas quando este não for seguido de uma expressão e for esperado um valor de retorno. 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.

Notas adicionais:

- `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`.

Desenvolvimento 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.

Submissão

O trabalho deverá ser validado no mooshak, usando o concurso criado especificamente para o efeito em `https://mooshak2.dei.uc.pt/~comp2017`. Como nas fases anteriores, 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 construção das

tabelas de símbolos e à deteção de erros de semântica, 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.

Fase IV – Geração de código intermédio

O gerador de código intermédio deve ser implementado em C utilizando as ferramentas `lex` e `yacc` a partir do código desenvolvido nas metas anteriores. Deverá chamar-se `jac`, como anteriormente, ler o programa a compilar do `stdin`, e emitir para o `stdout` um programa na representação intermédia do LLVM (v3.8) que implemente a mesma funcionalidade que o programa de entrada.

Por exemplo, a invocação:

```
./jac < gcd2.ja > gcd2.ll ↵
```

deverá processar o programa `gcd2.ja` e escrever o código IR LLVM correspondente em `gcd2.ll`. Este poderá ser executado diretamente com parâmetros de entrada na linha de comandos:

```
lli-3.8 gcd2.ll 12 8 ↵
```

ou compilado e ligado com:

```
llc-3.8 gcd2.ll
clang-3.8 -o gcd2 gcd2.s
```

podendo o executável resultante ser então invocado a partir da linha de comandos:

```
./gcd2 12 8 ↵
```

Em qualquer dos casos, deverá ser impresso no ecrã o resultado:

```
4
```

Para efeitos de verificação, o compilador deve fornecer ainda as seguintes opções definidas nas metas anteriores:

`-1` : executa a análise lexical, reportando eventuais erros lexicais, e termina.

`-l` : executa a análise lexical, reportando os tokens encontrados e eventuais erros lexicais, e termina.

`-2` : executa a análise sintática, reportando eventuais erros lexicais ou sintáticos, e termina.

`-t` : executa a análise sintática, reportando eventuais erros lexicais ou sintáticos, imprime a árvore de sintaxe abstrata construída durante a análise sintática do programa (se não houver erros sintáticos), e termina.

`-3` : executa a análise semântica (se não houver erros sintáticos), reportando eventuais erros lexicais ou semânticos, e termina. Caso haja erros sintáticos o comportamento é idêntico ao da opção `-2`.

`-s` : executa a análise semântica (se não houver erros sintáticos), reportando eventuais erros lexicais ou semânticos, imprime o conteúdo da(s) tabela(s) de símbolos e a árvore de sintaxe abstrata anotada, e termina. Caso haja erros sintáticos o comportamento é idêntico ao da opção `-2`.

Só deverá ser gerado código caso não haja erros de qualquer tipo (lexicais, sintáticos ou semânticos) nem sejam passadas quaisquer opções na linha de comandos.

Questões de implementação

Os tipos de dados boolean, int e double da linguagem Ja deverão ser implementados como os tipos i1, i32 e double da representação intermédia LLVM. Valores booleanos deverão ser impressos pelo comando System.out.println como false e true, enquanto valores inteiros e reais deverão ser impressos nos formatos "%d" e "%.16E" da função printf da linguagem C, respetivamente. Notar que em Java a formatação de reais é muito mais elaborada, conforme a documentação aplicável em https://docs.oracle.com/javase/8/docs/api/java/lang/Double.html#toString-double- detalha.

Submissão do trabalho

O trabalho deverá ser validado no mooshak, usando o concurso criado especificamente para o efeito em `https://mooshak2.dei.uc.pt/~comp2017`. Como nas fases anteriores, será tida em conta apenas a última submissão ao problema A desse concurso. Os restantes problemas destinam-se a ajudar na validação do gerador de código.

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.

Entrega final

Relembra-se que 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 o ficheiro `grupo.txt` com a identificação do grupo, no formato utilizado aquando da constituição dos grupos. 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.

ATENÇÃO: Apenas serão considerados para avaliação, discussão e defesa os projetos previamente avaliados no mooshak e submetidos no inforestudante!