```
void projecto_compiladores:
    print("
```

scurry

the language for ‘in-a-hurry’ programmers

```
    ");
end
```

```
main:
    #  David Marquês Francisco    José António Capela Dias
    #  nº 2007183509              nº 2007183794

    int semestre = 2;
    int ano = 2009-2010;

    #  Departamento de Engenharia Informática
    #  Faculdade de Ciências e Tecnologia da Univ. de Coimbra
end
```

Índice

Introdução

O projecto descrito neste relatório consiste num compilador de uma linguagem de programação procedimental com inspiração nas linguagens *Java*, *Ruby* e *Python*. Esta foi desenvolvida de forma a ter uma sintaxe intuitiva e o mais próxima quanto possível da linguagem natural. Possui todas as funcionalidades usuais das linguagens procedimentais incluindo tipos de dados básicos, declarações de variáveis, de procedimentos e funções, instruções de atribuição, de controlo, etc. A esta linguagem foi dado o nome de *Scurry*.

Descrição Geral do Projecto

De acordo com o que era pretendido, a linguagem desenvolvida possui vários tipos de dados e operações sobre os mesmos. Segue-se uma descrição das suas várias componentes.

Tipos de dados

Relativamente a tipos de dados numéricos, existem números inteiros, identificados pela palavra reservada *int*, e reais, declarados como *real*. Existe ainda um tipo de dados booleano *bool*, ao qual podem estar associados os valores de verdade *true* e *false*. Por último, fazem também parte da linguagem os tipos *char* e *string*, que representam caracteres e cadeias de caracteres, respectivamente.

Declaração de variáveis e atribuição

A declaração de variáveis é feita indicando o tipo da variável, seguido do identificador da mesma, da mesma forma que é feito nos tipos primitivos da linguagem *Java* ou *C*.

```
int my_var;
real pi;
char character;
bool my_bool;
```

Obviamente, é possível atribuir valores às variáveis declaradas, seja aquando da sua declaração ou numa altura posterior. Para isso é utilizado o operador de atribuição '=' seguido do valor desejado.

```
int zero = 0;
pi = 3.14159265;
character = 'c';
bool isAdmin = true;
```

É também possível atribuir a uma variável o valor de outra previamente declarada:

```
bool b0 = true;
bool b1 = !b0;
```

Procedimentos e funções

A definição de procedimentos segue a seguinte estrutura:

- No cabeçalho tem que estar presente o tipo de dados que a função devolve, seguido do nome da função. Os parâmetros, caso existam, são listados da seguinte forma:

```
(tipo1 identificador1, tipo2 identificador2, ...)
```

  Por fim, no cabeçalho da função é necessário colocar o caracter ":".
- De seguida, colocam-se as várias instruções referentes a essa função. Caso a função devolva um valor, esse valor deve ser guardado na variável *result*. Essa variável é declarada internamente com o tipo de retorno da função, sendo uma palavra reservada da linguagem.
- No final, a declaração é terminada com o *token* "*end*".

Assim, dois exemplos de declarações de procedimentos são:

```
void hello_world:
    println('Hello world');
end

void inutil(real r):
    r = 1.5;
    print('Valor de r igual a: ');
    println(r);
end
```

Já a declaração de funções envolve a utilização da variável *result* e da definição do tipo de dados a devolver no cabeçalho da função. Assim, um exemplo de uma declaração de função seria:

```
# Calcula a função de fibonacci de um número recebido por parâmetro
int fibonacci(int f):
    if f == 0:
        result = 0;
    else if f == 1:
        result = 1;
    else:
        result = fibonacci(f-2) + fibonacci(f-1);
    end
end
```

É obrigatória a existência do procedimento *main* (tal como nas linguagens *Java* e *C*) para que o programa seja executado. Este procedimento deve ser o último declarado no ficheiro e deve ter a seguinte forma:

```
main:
    ... # Instruções
end
```

Operações

Relativamente a operações **aritméticas**, estão presentes nesta linguagem as operações de adição (‘+’), subtracção (‘-’), divisão (‘/’), multiplicação (‘*’) e resto da divisão (‘mod’ ou ‘%’). Estas operações são válidas para os tipos de dados *integer* e *real*.

```
int a = 2;
a = a + (6/2 mod 1) - 15;

real radius = 3.5;
real pi = 3.14159265;
real perimeter = 2.0*pi*radius;
```

Relativamente às operações **relacionais**, estão presentes nesta linguagem as operações de igualdade (‘==’), maior (‘>‘), menor (‘<’), maior ou igual (‘>=’), menor ou igual (‘<=’), diferença (‘<>’). Estas operações são válidas para os tipos de dados *integer* e *real*. São ainda válidas com *char*, nas quais são comparados os seus valores *ascii*.

```
int a = 1;
int b = 2;

if a == b:
    println(‘Valor de a igual ao de b’);
else if a < b:
    println(‘Valor de a menor ao de b’);
end

bool c = a <> b;
```

Relativamente às operações **lógicas**, estão presentes nesta linguagem as operações de "e" lógico ('and'), "ou" lógico ('or') e a negação ('not' ou '!').

```
int a = 1;
int b = 2;

if a == 1 and b == 2:
   println('Valor de a igual a 1 e b igual a 2');
else if a == 1 or b == 2:
   println('Valor de a igual a 1 ou b igual a 2');
end

if not a == !a:
   # Verifica-se sempre
else:
   # Não entra aqui
end

bool c = not b == a;
```

Instruções de controlo

Relativamente às instruções **de controlo** existem dois tipos: de selecção e de repetição. As instruções de selecção presentes nesta linguagem são: *if*, *else* e *unless*. A estrutura da instrução *if-else* é semelhante à existente na linguagem *Python*, com a diferença dos *token else if* e *end*. Segue-se um exemplo da sua utilização:

```
# Calcula a função de fibonacci de um número recebido por parâmetro
int fibonacci(int f):
   if f == 0:
       result = 0;
   else if f == 1:
       result = 1;
   else:
       result = fibonacci(f-2) + fibonacci(f-1);
   end
end
```

A instrução '*unless*' permite que certas instruções só sejam executadas se a cláusula presente na instrução '*unless'* não se verificar. Um exemplo da sua aplicação:

```
x = 4 unless not admin;
```

Existe também a possibilidade de executar uma série de instruções dependendo da verificação ou não da condição definida na instrução *'unless'*. Para tal utiliza-se o *token 'do'* seguido de *':'*. De seguida, colocam-se as várias instruções e no final a cláusula '*unless*'. Um exemplo da sua utilização é:

```
do:
    print("Admin");
    println("istrador");
unless not admin;
```

As instruções de repetição presentes nesta linguagem são: *'for'*, *'while'*, *'do'*. A cláusula *'do until'* permite que se execute as instruções que estão no interior desta cláusula até que se verifique a condição pretendida. No entanto, este ciclo irá ser executado pelo menos uma vez pois a condição só é verificada no final.

```
do:
    a = a + 1;
    if a == 5: x = true;
until x == true;
```

Relativamente à cláusula *'for'* existem duas possibilidades: *'for downto'* e '*for to'*. A primeira refere-se a um ciclo no qual o valor da variável irá ser decrementado. Já a segunda refere-se ao caso em que o valor da variável será incrementado. Pode ainda especificar-se o valor do incremento. Seguem-se dois exemplos:

```
for a = 10 downto 1:
    a = a / 2;
end
```

```
for a = 1 to 10, 2: # Se não especificado, valor de incremento é 1
    a = a * 2;
end
```

Um exemplo de aplicação mais prático na realidade:

```
# Devolve o valor correspondente a base ^ exp
int pow(int base, int exp):
    result = 1;
    for a = 1 to exp:
        result = result * base;
    end
end
```

Por fim, a cláusula *'while'* permite que se execute as instruções que estão no interior desta cláusula até que se verifique a condição pretendida. Esta cláusula é semelhante à *'do until'* mas no entanto, este ciclo não obriga a que as instruções sejam executadas pelo menos uma vez pois a condição é verificada logo no início. Um exemplo da sua utilização é:

```
while a < 8:
    a = a + 1;
end
```

Instrução de output

A instrução de **output** é feita através dos *tokens 'print'* e *'println'*. O argumento recebido por estas funções pode ser uma cadeia de caracteres, um inteiro, um número real, um booleano ou uma variável. Seguem-se alguns exemplos:

```
print("Administrador");
print(0 + 1);
println('0' + '1');
println(a);
```

Expressões condicionais

As expressões condicionais são descritas utilizando o *token '?'*. Estas expressões oferecem uma alternativa à utilização da cláusula *'if-else'*. Seguem-se dois exemplos:

```
not admin? print("Not admin");
admin? print("Admin") : print("Not admin"); # Semelhante à linguagem C
```

Comentários

A adição de comentários é feita introduzindo o token '#'. Os comentários podem existir numa linha isolada ou depois de determinada instrução. Seguem-se dois exemplos da sua utilização:

```
println((5*10)%3); # % equivale a mod
# Uma linha apenas com um comentário
```

Implementação das estruturas

Análise lexical e sintáctica

No que diz respeito à implementação desta linguagem, foi utilizado o módulo *PLY* (*Python Lex-Yacc*) que é uma implementação das ferramentas de *parsing lex* e *yacc* para a linguagem de programação *Python*. Esta implementação suporta *parsing LALR(1)* e também possibilita validação extensiva de *input*, tratamento de erros e diagnósticos.

No que toca à organização, está dividida em dois módulos: *lex.py* e *yacc.py*. O módulo *lex* permite criar um analisador lexical, ou seja, dividir o *input* em *tokens* de acordo com expressões regulares previamente definidas. O módulo *yacc* permite criar a gramática referente à nossa linguagem através da definição de uma série de regras. O *yacc* recebe do *lex* os *tokens* e processa-os de acordo com a gramática definida, gerando tabelas de *parsing LALR(1)*. Os *tokens* e as regras que serão utilizados por esses módulos estão definidos nos ficheiros "*tokens.py*" e "*rules.py*", respectivamente.

Sintaxe abstracta

O ficheiro *yacc.py* do módulo *PLY* não oferece qualquer mecanismo que permita a construção da árvore de sintaxe abstracta. No entanto, em *Python* é simples criar uma árvore deste tipo (que possa ter um conjunto de filhos de vários tipos e em número variável). Existem várias abordagens possíveis – pode ser criado um tipo de nó para sensivelmente cada tipo de produção ou então criar uma classe genérica que suporte qualquer número de argumentos. Esta última foi a abordagem seguida para a criação do *Scurry*.

```
# Representa os nós da árvore de sintaxe abstracta
class Node(object):
    def __init__(self, t, lineno, *args):
        self.type = t
        self.lineno = lineno
        self.args = args

    def __str__(self): # Apenas para debugging
        ''' String que representa a informação do nó '''
```

```
        s = "type: " + str(self.type) + "\n"
        s += "".join(["i: " + str(i) + "\n" for i in self.args])
        return s

    @staticmethod
    def is_node(n):
        ''' Devolve se o parametro é ou não um objecto Node '''
        return type(n) == type(Node("", 0))
```

A especificação da sintaxe abstracta é feita posteriormente neste documento.

Análise semântica

Tal como acontece para a árvore de sintaxe abstracta, a análise semântica é também feita sem recurso a módulos que simplifiquem o processo. Criaram-se duas classes – uma que representa um contexto, e uma que implementa uma pilha de contextos.

```
# Tabelas de símbolos
# Ambientes que mapeiam identificadores com os seus tipos e localizações
class Context(object):
    def __init__(self,name=None):
        self.variables = {}
        self.var_count = {}
        self.name = name

    def has_var(self,name):
        return name in self.variables

    def get_var(self,name):
        return self.variables[name]

    def set_var(self,typ,name):
        self.variables[name] = typ
        self.var_count[name] = 0

class ContextStack(object):
    def __init__(self):
        self.contexts = []

    def pop(self):
        count = self.contexts[-1].var_count
        for v in count:
            if count[v] == 0:
                warning("variable %s was declared, but not used." % v)
        self.contexts.pop()
```

```
    def push(self, obj):
        self.contexts.append(obj)

    def top(self):
        return self.contexts[-1]

    def __iter__(self):
        return iter(self.contexts[::-1])

context_stack = ContextStack() # Pilha global
```

Geração de código

A geração de código possui o auxílio do módulo *PY-LLVM*. Este oferece várias primitivas que nos permitiram criar vários modos de utilização do nosso compilador:

- Geração do executável final;
- Geração e execução do executável final;
- Geração de código *assembly*;
- Geração de código *C reduzido*;

Apresenta-se de seguida uma pequena porção de código relacionada com esses modos:

```
if options.run: # Executar o programa em vez de o gravar em ficheiro
      from llvm.core import _core
      bytecode = _core.LLVMGetBitcodeFromModule(o.ptr)

      p = Popen(['lli'],stdout=PIPE, stdin=PIPE)
      sys.stdout.write(p.communicate(bytecode)[0])

elif options.cfile: # Gerar o ficheiro em c√≥digo C reduzido
        o.to_bitcode(file("tmp/middle.bc", "w"))
        os.system("llc tmp/middle.bc -o %s -march c"  % options.cfile)

else: # Gerar o ficheiro execut√°vel final
      o.to_bitcode(file("tmp/middle.bc", "w"))
      os.system("llc -f -o=tmp/middle.s tmp/middle.bc")
      os.system("gcc -o %s tmp/middle.s" % options.filename)
```

Especificação da Gramática

Palavras reservadas e outros símbolos terminais

Segue-se a lista de palavras reservadas desta gramática:

```
   'main':    'MAIN',
   'end':     'END',

# Controlo de fluxo de execução
   'if':      'IF',
   'else':    'ELSE',
   'unless':  'UNLESS',
   'for':     'FOR',
   'while':   'WHILE',
   'do':      'DO',
   'to':      'TO',
   'downto':  'DOWNTO',
   'until':   'UNTIL',

# Expressões lógicas
   'and':    'AND',
   'or':     'OR',
```

```
   'true':    'TRUE',
   'false':   'FALSE',
   'global':  'GLOBAL',

# Funções
   'void':  'VOID',

# Nomes dos tipos de dados
   'real':    'TREAL',
   'int':     'TINT',
   'string':  'TSTRING',
   'char':    'TCHAR',
   'bool':    'TBOOL'
   'not':     'NOT',
   'mod':     'MOD'
```

Lista de *tokens*:

```
# Declarações
t_IDENTIFIER = r'[A-z][A-z0-9_]*'
t_ASSIGNMENT = r"="
t_SEMICOLON  = r";"
t_COLON      = r":"
t_COMMA      = r","
t_QMARK      = r'\?'

# Comparadores lógicos
t_EQ         = r"\=\="
t_NEQ        = r"\<\>"
t_LT         = r"\<"
t_GT         = r"\>"
t_LTE        = r"\<\="
t_GTE        = r"\>\="
t_NOT        = r"\!"
```

```
# Operadores matemáticos
t_PLUS       = r"\+"
t_MINUS      = r"\-"
t_TIMES      = r"\*"
t_DIVISION   = r"/"
t_MOD        = r"\%"
t_POW        = r"\*\*"

# Funções e operações matemáticas
t_LPAREN     = r"\("
t_RPAREN     = r"\)"
```

```
  # Tipos de dados
  t_REAL        = r"[0-9]+\.[0-9]+"
  t_INT         = r"[0-9]+"
  t_CHAR        = r"(\'([^\\\']|\')|(\"([^\\\"]|\")"
  t_STRING      = "(\"([^\\\"]|(\\.))*\")|(\'([^\\\']|(\\.))*\')"

  # Nontokens
  t_COMMENT = r"[ ]*\043[^\n]*"  # Ignorar conteúdo dos comentários. \043 é #
  t_newline = r'\n+' # Regra para guardar o nº da linha em que estamos
  t_ignore  = ' \t'   # Caracteres a ignorar (espaços e tabs)
```

Estabelecimento de precedências

Tal como referido anteriormente, as regras da gramática encontram-se no ficheiro "rules.py". Neste ficheiro, começam-se por definir as prioridades das operações:

```
precedence = (
    # Prioridades ao nível das operações matemáticas
    ('left', 'PLUS', 'MINUS'),
    ('left', 'TIMES', 'DIVISION', 'MOD'),
    # Operações lógicas
    ('left', 'EQ', 'NEQ', 'LTE','LT','GT','GTE'),
    ('left', 'OR', 'AND'),
)
```

Produção base

Após serem descritas as precedências, definem-se as regras gramaticais que serão utilizadas. A regra de início da gramática é descrita por:

```
def p_program(p):
      'program : proc_func_and_var MAIN COLON statement_sequence END'
      p[0] = Node('program',ln(p),p[1],p[4])
```

Que corresponde, na notação convencional das gramáticas, a:

```
program → proc_func_and_var MAIN COLON statement_sequence END
```

Posteriormente, as produções da gramática serão sempre descritas dessa forma.

Declaração de variáveis e funções

Antes da função *main*, que é o último procedimento a ser especificado, um programa em *Scurry* pode iniciar-se com uma declaração de uma variável global, de uma função ou de um procedimento:

```
proc_func_and_var → globalvar_declaration proc_func_and_var
                  | procedure_declaration proc_func_and_var
                  | function_declaration  proc_func_and_var
                  |
```

A produção vazia serve para o caso em que apenas existe a função *main*, sem qualquer procedimento, função ou variável global adicional. Seguem-se as regras referentes à declaração e atribuição de variáveis globais e locais:

```
globalvar_declaration → GLOBAL type identifier SEMICOLON

variable_declaration → type identifier

assignment_statement → identifier ASSIGNMENT expression

declaration_and_assignment → type identifier ASSIGNMENT expression
```

As regras que se referem aos tipos e identificadores de variáveis são as seguintes:

```
identifier → IDENTIFIER

type → TREAL
     | TINT
     | TCHAR
     | TBOOL
     | TSTRING

real → REAL
     | MINUS REAL

int → INT
    | MINUS INT

string → STRING

char → CHAR

bool → TRUE
     | FALSE
```

As produções que se seguem destinam-se à declaração de funções e procedimentos, no que diz respeito aos seus cabeçalhos e corpo.

```
procedure_declaration → procedure_heading COLON statement_sequence END

procedure_heading → VOID identifier
                  | VOID identifier LPAREN parameter_list RPAREN

function_declaration → function_heading COLON statement_sequence END

function_heading → type identifier
                 | type identifier LPAREN parameter_list RPAREN
```

Como se pode verificar, os procedimentos diferem das funções pelo facto de possuírem no início do seu cabeçalho a palavra reservada *void* em vez do seu tipo de retorno. No que diz respeito à passagem de parâmetros:

```
parameter_list → parameter COMMA parameter_list
               | parameter

parameter → type identifier
```

A passagem de parâmetros é sempre feita por valor. É também necessário definir as regras para as chamadas de funções e indicação de parâmetros:

```
procedure_or_function_call → identifier LPAREN param_list RPAREN
                           | identifier

function_call_inline → identifier LPAREN param_list RPAREN

param_list → param_list COMMA param
           | param

param → expression
```

Sequências e outras instruções

Nesta secção, definem-se as restantes regras existentes. As suas funções são auto-explicativas.

```
statement_sequence → statement statement_sequence
                   |

statement → assignment_statement SEMICOLON
          | if_statement
          | conditional_expression SEMICOLON
          | do_statement SEMICOLON
          | while_statement
          | for_statement
```

```
          | procedure_or_function_call SEMICOLON
          | variable_declaration SEMICOLON
          | declaration_and_assignment SEMICOLON
          | unless_statement SEMICOLON

if_statement → IF expression COLON statement_sequence ELSE COLON
               statement_sequence END
             | IF expression COLON statement_sequence END
             | IF expression COLON statement_sequence ELSE if_statement

conditional_expression → expression QMARK statement_no_semicolon
                       | expression QMARK statement_no_semicolon COLON
                         statement_no_semicolon

unless_statement → statement_no_semicolon UNLESS expression
                 | DO COLON statement_sequence UNLESS expression

statement_no_semicolon → assignment_statement
                       | procedure_or_function_call
                       | variable_declaration
                       | declaration_and_assignment

while_statement → WHILE expression COLON statement_sequence END

do_statement → DO COLON statement_sequence UNTIL expression

for_statement → FOR assignment_statement TO expression COLON
                statement_sequence END
              | FOR assignment_statement TO expression COMMA expression
                COLON statement_sequence END
              | FOR assignment_statement DOWNTO expression COLON
                statement_sequence END
              | FOR assignment_statement DOWNTO expression COMMA
                expression COLON statement_sequence END

expression → expression and_or expression_m
           | expression_m

expression_m → element
             | element sign expression_m

and_or → AND
       | OR

sign → PLUS
     | MINUS
     | DIVISION
     | TIMES
     | MOD
     | POW
```

```
      | EQ
      | NEQ
      | LT
      | LTE
      | GT
      | GTE

element → identifier
        | real
        | int
        | string
        | char
        | bool
        | LPAREN expression RPAREN
        | NOT element
        | function_call_inline
```

Tratamento de erros

O tratamento de erros é feito ao nível da análise lexical, sintáctica, semântica e na geração de código.

No que diz respeito à **análise lexical**, por *default*, o programa *lex.py* do pacote *PLY* não tem qualquer conhecimento posicional, isto é, sobre em que linha e coluna terá ocorrido o erro. Isto acontece pelo simples facto de sermos nós a definir o que é uma linha. No nosso caso, definimos por isso um *token newline* que, ao ser encontrado no código de *input*, incrementa um atributo *lineno* que o *lexer* possui para facilitar este processo. Por outro lado, não possui igualmente informação sobre a coluna em que nos encontramos. No entanto, possui uma variável *lexpos* actualizada em cada novo *token* e que nos permite saber o número de caracteres consumidos desde o início da sua execução. Com base nestas duas variáveis, o compilador *Scurry* indica a linha e coluna de qualquer erro lexical.

Uma vez que é conveniente que o *lexer* não termine quando encontra um erro, é feito um `t.lexer.skip(1)` que ignora o caracter erróneo encontrado e continua a sua análise.

```
# Nontoken - Regra para guardar o número da linha em que nos encontramos
def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)
    global lexpos
    lexpos = t.lexpos # Armazenar número de caracteres já consumidos

...
```

```
def t_error(t):
    ''' Regra para tratamento de erros '''
    print "Error [line %d column %d]: Illegal character '%s'" %(t.lineno,\
        find_column(t), t.value[0])
    t.lexer.skip(1)

lexpos = 0 # Para armazenar número de caracteres consumidos até ao momento

def find_column(t):
    ''' Identificar a coluna em que ocorreu o erro '''
    # token.lexpos devolve o número de caracteres desde o inicio
    # Nós queremos o número de caracteres desde o princípio desta linha
       return t.lexpos-lexpos
```

No que diz respeito à **análise sintáctica**, a documentação do *PLY* refere que "*error recovery in LR parsers is a delicate topic that involves ancient rituals and black-magic*". Não é por isso simples recuperar de erros sintácticos. A abordagem que seguimos para o compilador *Scurry* é, após imprimir a mensagem de erro, prosseguir a análise ignorando o erro encontrado. Esta abordagem nem sempre resulta, ilustrando por vezes erros sintácticos após o primeiro erro que não fazem sentido e que decorrem deste primeiro erro encontrado.

Uma vez que na análise sintáctica já se está a trabalhar sobre as *tokens*, e não carácter a carácter, as mensagens de erro e de aviso mostradas ao utilizador apenas possuem informação sobre a linha (e não sobre a coluna como acontece na análise lexical).

```
# Error rule for syntax errors
def p_error(p):
    global last_error_msg

    try: error_msg = "Syntax error [line %d]: Erroneous input" %p.lineno
    except: error_msg = default_msg

    if error_msg == last_error_msg or (last_error_msg and error_msg == \
       default_msg):
        sys.exit()
    print error_msg
    last_error_msg = error_msg # Guardar a última mensagem de erro. Isto é
    # feito uma vez que o comportamento de yacc.errok() nem sempre é o
    # esperado, e pode entrar em ciclo ilustrando sempre a mesma mensagem de
    # erro
    yacc.errok() # Tenta continuar apesar do erro que ocorreu
    return yacc.token() # Devolve o próximo token da stream
```

Uma vez terminada a análise sintáctica, sabemos se as expressões de entrada fornecidas obedecem às regras da gramática da nossa linguagem. No entanto, existem muitas verificações que não podem ser feitas a esta nível, como por exemplo, utilização de variáveis não declaradas, incompatibilidade de tipos, entre outros. Este tipo de erros tem de ser verificado caso a caso aquando da análise semântica.

Segue-se um exemplo de uma verificação semântica para o caso das instruções lógicas *and* e *or*:

```
...
elif node.type == 'and_or':
   op = node.args[0].args[0]
   for i in range(1,2):
      a = check(node.args[i])
      if a != "bool":
          raise_exception(node,"%s requires a boolean. Got %s instead." %(op,a))

...
```

Para se ter informação sobre a linha em que ocorreu o erro, esta é passada para o nó aquando da criação da árvore de sintaxe abstracta, na análise sintáctica.

```
def raise_exception(node, message):
   raise Exception, "Semantic error [line %d]: %s" %(node.lineno, message)

def raise_nolineno(message):
   raise Exception, "Semantic error: %s" % message

def warning(message):
      print "Warning:", message
```

Especificação da Sintaxe Abstracta

No ficheiro *ast.py* encontra-se a classe *Node* que representa os nós da árvore de sintaxe abstracta. Cada nó irá ter como atributo o seu tipo, o número da linha em que a regra foi encontrada e um número variável de argumentos. Esta informação é criada na análise sintáctica e utilizada na análise semântica, no ficheiro *semantic.py*. Cada tipo de nó é verificado de acordo com as suas características. Segue-se uma porção do código do ficheiro *semantic.py*.

```
def check(node):
   if not Node.is_node(node):
       # Se for uma lista de nós (é iterable mas não é string)
       if hasattr(node,"__iter__") and type(node) != type(""):
           for i in node:
               check(i)
       else:
           return node
   else:
       current_node = node

       if node.type in ["program"]:
           context_stack.push(Context())
           check(node.args)
           context_stack.pop()

       elif node.type in ['identifier']:
           return node.args[0]

       elif node.type in ['proc_func_and_var_list','statement_list']:
           return check(node.args)
...
```

Para finalizar este documento, segue-se a especificação da sintaxe abstracta.

```
program → <proc_func_and_var_list><statement_list>

# Declaração e atribuição de variáveis

proc_func_and_var_list → <global_var><proc_func_and_var_list>
                       v <procedure><proc_func_and_var_list>
                       v <function><proc_func_and_var_list>

global_var → <type><identifier>
```

```
var → <type><identifier>

assign → <identifier><op>

var_assign → <type><identifier><op>

# Nomes e tipos de variáveis

identifier → IDENTIFIER

type → TREAL ∨ TINT ∨ TCHAR ∨ TBOOL ∨ TSTRING

real → REAL

int → INT

string → STRING

char → CHAR

bool → TRUE ∨ FALSE

# Funções com e sem valor de retorno

procedure → <procedure_head><statement_list>

procedure_head → <identifier>
               ∨ <identifier><parameter_list>

function → <function_head><statement_list>

function_head → <type><identifier>
              ∨ <type><identifier><parameter_list>

# Listas de parâmetros

parameter_list → <parameter><parameter_list>
               ∨ <parameter>

parameter → <type><identifier>

# Chamada de funções e passagem de parâmetros

function_call → <identifier>
              ∨ <identifier><parameter_list>

function_call_inline → <identifier><parameter_list>

parameter_list → <parameter>
               ∨ <parameter_list><parameter>

parameter → <expression>

# Sequências e instruções de repetição

statement_list  → <assign><statement_list>
                ∨ <if><statement_list>
                ∨ <do><statement_list>
                ∨ <while><statement_list>
                ∨ <for><statement_list>
```

```
                ∨ <function_call><statement_list>
                ∨ <var><statement_list>
                ∨ <var_assign><statement_list>
                ∨ <unless><statement_list>

if  → <op><statement_list><statement_list> # If - Else
    ∨ <op><statement_list> # If
    ∨ <op><statement_list><if> # If - Else if
    ∨ <op><assign> # x ? y (expressões condicionais são convertidas em If)
    ∨ <op><function_call>
    ∨ <op><var>
    ∨ <op><var_assign>
    ∨ <op><assign><statement_list> # x ? y : z
    ∨ <op><function_call><statement_list>
    ∨ <op><var><statement_list>
    ∨ <op><var_assign><statement_list>

unless → <assign><op> # x unless y
       ∨ <function_call><op>
       ∨ <var><op>
       ∨ <var_assign><op>
       ∨ <statement_list><op>

while → <op><statement_list>

do → <statement_list><op>

for → <assign><TO><op><statement_list> # for a = 1 to 10: ... end
    ∨ <assign><TO><op><op><statement_list> # for a = 1 to 10, 1: ... end
    ∨ <assign><DOWNTO><op><statement_list> # for a = 1 downto 10: ... end
    ∨ <assign><DOWNTO><op><op><statement_list>

op → <op><and_or><op> # expression and_or expression_m
   ∨ <op>
   ∨ <element>
   ∨ <element><sign><op> # expression sign expression_m

and_or → AND ∨ OR

sign → PLUS ∨ MINUS ∨ DIVISION ∨ TIMES ∨ MOD ∨ POW ∨ EQ ∨ NEQ
     ∨ LT ∨ LTE ∨ GT ∨ GTE

element → <identifier>
        ∨ <real>
        ∨ <int>
        ∨ <string>
        ∨ <char>
        ∨ <bool>
        ∨ <function_call>

not → <element>
element → <op>
```

Extras

Desenvolveram-se algumas funcionalidades que não faziam parte do pedido no enunciado. Segue-se a lista dos extras presentes neste projecto:

- Conversão do código-fonte para *html* e *rtf*, com *highlighted syntax*, utilizando um tipo de *parser* diferente do *lex*;
- Recursividade;
- Variáveis globais (se bem que possui alguns *bugs*);
- Expressões condicionais;
- Opção de geração de código *Assembly* ou de código *C* reduzido;

Conclusão

O presente trabalho permitiu-nos fundamentalmente aplicar conceitos e conhecimentos teóricos previamente adquiridos na cadeira. Os compiladores são provavelmente as ferramentas mais importantes no mundo da informática, e aprender a desenvolver um compilador e uma linguagem de programação permitiu-nos compreender melhor como é que tudo se processa a um nível mais baixo. Para além disso, apesar de já existir uma imensa quantidade de linguagens, é sempre útil aprender a criá-las, uma vez que é muitas vezes vantajoso criar linguagens específicas para problemas concretos. Por isso, o desenvolvimento deste projecto forneceu-nos competências importantes para a nossa formação académica. Desta forma, consideramos que o trabalho foi muito interessante e simultaneamente importante.

Autores

David Marquês Francisco nº 2007183509

José António Capela Dias nº 2007183794

Departamento de Engenharia Informática da Faculdade de Ciências e Tecnologia

Universidade de Coimbra, 2009/2010