

### 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 UC.

#### 3.1 Gramática inicial em notação EBNF

FunctionsAndDeclarations $\rightarrow$ (FunctionDefinition   FunctionDeclaration   Declaration) {FunctionDefinition   FunctionDeclaration   Declaration}
FunctionDefinition $\rightarrow$ TypeSpec FunctionDeclarator FunctionBody
FunctionBody $\rightarrow$ LBRACE [DeclarationsAndStatements] RBRACE
DeclarationsAndStatements $\rightarrow$ Statement DeclarationsAndStatements   Declaration DeclarationsAndStatements   Statement   Declaration
FunctionDeclaration $\rightarrow$ TypeSpec FunctionDeclarator SEMI
FunctionDeclarator $\rightarrow$ ID LPAR ParameterList RPAR
ParameterList $\rightarrow$ ParameterDeclaration {COMMA ParameterDeclaration}
ParameterDeclaration $\rightarrow$ TypeSpec [ID]
Declaration $\rightarrow$ TypeSpec Declarator {COMMA Declarator} SEMI
TypeSpec $\rightarrow$ CHAR   INT   VOID   SHORT   DOUBLE
Declarator $\rightarrow$ ID [ASSIGN Expr]
Statement $\rightarrow$ [Expr] SEMI
Statement $\rightarrow$ LBRACE {Statement} RBRACE
Statement $\rightarrow$ IF LPAR Expr RPAR Statement [ELSE Statement]
Statement $\rightarrow$ WHILE LPAR Expr RPAR Statement
Statement $\rightarrow$ RETURN [Expr] SEMI
Expr $\rightarrow$ Expr (ASSIGN   COMMA) Expr
Expr $\rightarrow$ Expr (PLUS   MINUS   MUL   DIV   MOD) Expr
Expr $\rightarrow$ Expr (OR   AND   BITWISEAND   BITWISEOR   BITWISEXOR) Expr
Expr $\rightarrow$ Expr (EQ   NE   LE   GE   LT   GT) Expr
Expr $\rightarrow$ (PLUS   MINUS   NOT) Expr
Expr $\rightarrow$ ID LPAR [Expr {COMMA Expr}] RPAR
Expr $\rightarrow$ ID   INTLIT   CHRLIT   REALLIT   LPAR Expr RPAR

Uma vez que a gramática dada é ambígua e é apresentada em notação EBNF, onde [...] representa “opcional” e {...} representa “zero ou mais repetições”, esta deverá ser modificada para permitir a análise sintática ascendente com o yacc. Será necessário ter em conta a precedência e as regras de associação dos operadores, entre outros aspetos, de modo a garantir a compatibilidade entre as linguagens UC e C. Note que o operador COMMA é associativo à esquerda.

### 3.2 Programação do analisador

O analisador deverá chamar-se *ucompiler*, 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 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 e de coluna, e pelo valor semântico do token que dá origem ao erro. Isto pode ser conseguido definindo a função:

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

A analisador deve ainda incluir 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):

Declaration → error SEMI  
Statement → error SEMI  
Statement → LBRACE error RBRACE  
Expression → ID LPAR error RPAR  
Expression → LPAR error RPAR

### 3.4 Árvore de sintaxe abstrata (AST)

Caso seja feita a seguinte invocação:

```
./uccompiler -t < first.uc
```

deverá gerar a árvore de sintaxe abstrata correspondente, e imprimi-la no stdout de acordo com a especificação 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 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 ( $\geq 1$ ) (<variable and/or function declarations>)

### Declaração de variáveis

Declaration ( $\geq 2$ ) (<typespec> Id)

### Declaração/definição de Funções

FuncDeclaration (3) (<typespec> Id ParamList)

FuncDefinition (4) (<typespec> Id ParamList FuncBody)

ParamList ( $\geq 1$ ) (ParamDeclaration)

FuncBody ( $\geq 0$ ) (<declarations> | <statements>)

ParamDeclaration( $\geq 1$ ) (<typespec> [Id])

### Statements

StatList( $\geq 2$ ) If(3) While(2) Return(1)

### Operadores

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)

Not(1) Minus(1) Plus(1) Store(2) Comma(2) Call( $\geq 1$ ) BitWiseAnd(2) BitWiseXor(2)

BitWiseOr(2)

### Terminais

Char, ChrLit, Id, Int, Short, IntLit, Double, RealLit, Void

### Especial

Null (na ausência de um nó filho obrigatório)

**Nota:** Não deverão ser gerados nós supérfluos, nomeadamente StatList com menos de *statements* no seu interior. Os nós Program, ParamList e FuncBody não deverão ser considerados redundantes mesmo que tenham menos de dois nós filhos.

A Figura 2 exemplifica a impressão da árvore de sintaxe abstrata do programa apresentado na primeira página.

```
Program
..FuncDefinition
...Int
....Id(main)
....ParamList
.....ParamDeclaration
.....Void
...FuncBody
...
```

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

### 3.5 Desenvolvimento do analisador

Sugere-se que desenvolva o analisador de forma faseada. Deverá começar por re-escrever a gramática acima apresentada para o yacc de modo a permitir a deteção de eventuais erros de sintaxe. Após terminada esta fase, e já com garantia 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 elementos do grupo, sugere-se que comecem por analisar produções diferentes. Observando o não-terminal `FunctionsAndDeclarations`, um elemento começaria por `FunctionsAndDeclarations`  $\rightarrow$  `FunctionDefinition` {`FunctionDefinition`} enquanto o outro começaria por `FunctionsAndDeclarations`  $\rightarrow$  `Declaration` {`Declaration`}. 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 Submissão 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 *ucompiler.l* e *ucompiler.y* e ser colocados num único arquivo com o nome *ucompiler.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.