

Projeto de Compiladores 2017/18

Compilador para a linguagem UC

16 de março de 2018

Este projeto consiste no desenvolvimento de um compilador para a linguagem UC, que é um subconjunto da linguagem C (de acordo com o standard C99).

Na linguagem UC é possível usar variáveis e literais do tipo `char`, `short`, `int`, e `double` (todos com sinal). A linguagem UC inclui expressões aritméticas e lógicas, instruções de atribuição, operadores relacionais, e instruções de controlo (`if-else` e `while`). Inclui também funções com os tipos de dados já referidos, sendo a passagem de parâmetros sempre feita por valor. A ausência de parâmetros de entrada ou de valor de retorno é identificada pela palavra-chave `void`.

A função invocada no início de cada programa chama-se `main`, tem valor de retorno do tipo `int` e não recebe parâmetros, sendo que o programa `int main(void) { return 0; }` é um dos mais pequenos possíveis na linguagem UC. Os programas podem ler e escrever caracteres na consola através das funções pré-definidas `getchar()` e `putchar()`, respetivamente.

O significado de um programa na linguagem UC será o mesmo que em C99, assumindo a pré-definição das funções `getchar()` e `putchar()`. Por fim, são aceites comentários nas formas `/* ... */` e `// ...` que deverão ser ignorados. Assim, por exemplo, o programa que se segue imprime na consola os caracteres de A a Z:

```
int main(void) {
    char i = 'A';
    while (i <= 'Z') {
        putchar(i);
        i = i + 1;
    }
    return 0;
}
```

1 Metas e avaliação

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

1. Análise lexical (16%) – 1 de março de 2018
2. Análise sintática (26%) – 5 de abril de 2018
3. Análise semântica (26%) – 4 de maio de 2018
4. Geração de código (26%) – 30 de maio de 2018

A entrega final será acompanhada de um relatório com um peso de 6% na avaliação. O trabalho será obrigatoriamente verificado no MOOSHAK, em cada uma das metas, usando um concurso criado especificamente para o efeito. Para além disso, a entrega final do trabalho deverá ser feita através do InforEstudante, até às 23h59 do dia 30 de maio de 2018, e incluir todo o código-fonte produzido no âmbito do projeto (exatamente os mesmos .zip que tiverem sido colocados no MOOSHAK em cada meta) e um ficheiro grupo.txt contendo os dados do grupo, no formato:

```
Grupo: <nome_do_grupo>          ──>  Sugestão: username1_username2 das contas no DEI
Nome1: <nome_aluno_1>
Numero1: <numero_aluno_1>
Email1: <email_aluno_1>
Nome2: <nome_aluno_2>
Numero2: <numero_aluno_2>
Email2: <email_aluno_2>
```

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 e terá lugar entre os dias 4 e 15 de junho de 2018. A nota final do projeto diz respeito à prestação individual na defesa e está limitada pela soma ponderada das pontuações obtidas no MOOSHAK em cada uma das metas. Assim, a classificação final nunca poderá exceder a pontuação obtida no MOOSHAK acrescida da classificação do relatório final. Aplica-se mínimos de 47.5% à nota final após a defesa.

2 Meta 1 – Analisador lexical

Nesta primeira meta deve ser programado um analisador lexical para a linguagem UC. A programação deve ser feita recorrendo à linguagem de programação C utilizando a ferramenta *lex*. Os “tokens” a ser considerados pelo compilador deverão estar de acordo com o C99 standard¹ e são apresentados em seguida.

2.1 Tokens da linguagem UC

ID: sequências alfanuméricas começadas por uma letra, onde o símbolo “_” conta como uma letra. Letras maiúsculas e minúsculas são consideradas letras diferentes.

INTLIT: sequências de dígitos decimais (0–9).

CHRLIT: um único carácter (excepto *newline* ou aspa simples) ou uma “sequência de escape” entre aspas simples. Apenas as sequências de escape `\n`, `\t`, `\\`, `\'`, `\"` e `\ooo` são definidas pela linguagem, onde ooo representa uma sequência de 1 a 3 dígitos entre 0 e 7. A ocorrência de uma sequência de escape inválida ou de mais do que um carácter ou sequência de escape entre aspas simples deve dar origem a um erro lexical.

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

¹ISO C 1999 Standard – <https://tinyurl.com/comp2018>

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.

CHAR = char

ELSE = else

WHILE = while

IF = if

INT = int

SHORT = short

DOUBLE = double

RETURN = return

VOID = void

BITWISEAND = “&”

BITWISEOR = “|”

BITWISEXOR = “^”

AND = “&&”

ASSIGN = “=”

MUL = “*”

COMMA = “,”

DIV = “/”

EQ = “==”

GE = “>=”

GT = “>”

LBRACE = “{”

LE = “<=”

LPAR = “(”

LT = “<”

MINUS = “-”

MOD = “%”

NE = “!=”

NOT = “!”

OR = “||”

PLUS = “+”

RBRACE = “}”

RPAR = “)”

SEMI = “;”

RESERVED: palavras reservadas da linguagem C não utilizadas em UC, bem como os símbolos “[”, “]”, o operador de incremento (“++”) e o operador de decremento (“--”).

2.2 Programação do analisador

O analisador deverá chamar-se *uccompiler*, ler o ficheiro a processar através do *stdin* e, se invocado com a opção *-l*, emitir o resultado da análise lexical para o *stdout* e terminar. Na ausência de qualquer opção, deve escrever no *stdout* apenas as mensagens de erro. Caso o ficheiro *first.uc* contenha o programa de exemplo dado anteriormente, que imprime os caracteres de A a Z, a invocação:

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

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

```
INT
ID(main)
LPAR
VOID
RPAR
LBRACE
CHAR
ID(i)
ASSIGN
CHRLIT('A')
SEMI
WHILE
LPAR
```

```

ID(i)
LE
CHRLIT('Z')
RPAR
LBRACE
ID(putchar)
LPAR
ID(i)
RPAR
SEMI
ID(i)
ASSIGN
ID(i)
PLUS
INTLIT(1)
SEMI
RBRACE
RETURN
INTLIT(0)
SEMI
RBRACE

```

O analisador deve aceitar (e ignorar) como separador de tokens o espaço em branco (espaços, tabs e mudanças de linha), bem como comentários do tipo `/* ... */` 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 acima para ID e INTLIT.

2.3 Tratamento de erros

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

```

"Line <num linha>, col <num coluna>: invalid char constant (<c>)\n"
"Line <num linha>, col <num coluna>: unterminated comment\n"
"Line <num linha>, col <num coluna>: unterminated char constant\n"
"Line <num linha>, col <num coluna>: illegal character (<c>)\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. O analisador deve recuperar da ocorrência de erros lexicais a partir do *fim* desse token.

2.4 Submissão da meta 1

O trabalho deverá ser validado no MOOSHAK, usando o concurso criado especificamente para o efeito em <https://mooshak2.dei.uc.pt/~comp2018/>. Será tida em conta apenas a 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 depuração.

O ficheiro *lex* a submeter deverá chamar-se `ucompiler.1`, listar os autores num comentário e ser enviado num ficheiro com o nome `ucompiler.zip`, que não deverá ter quaisquer diretorias.

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 `uccompiler`, ler o ficheiro a processar através do `stdin` e emitir todos os resultados para o `stdout`. Por uma questão de compatibilidade com a fase anterior, se o analisador for invocado com a opção `-l`, 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 opção, o analisador deve detetar a existência de quaisquer erros lexicais e de sintaxe no ficheiro de entrada, e emitir as mensagens de erro correspondentes para o `stdout`.

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` 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) (<variable and/or function declarations>)

Declaração de variáveis

Declaration (2) (<typespec> Id)

Declaração/definição de Funções

FuncDeclaration (3) (<typespec> Id ParamList)

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

ParamList (≥ 1) (ParamDeclaration)

FuncBody (≥ 0) (<declarations> | <statements>)

ParamDeclaration(≥ 1) (<typespec> [Id])

Statements

StatList(≥ 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(≥ 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.

No caso do programa dado, o resultado deve ser:

```
Program
..FuncDefinition
....Int
....Id(main)
....ParamList
.....ParamDeclaration
.....Void
....FuncBody
.....Declaration
.....Char
.....Id(i)
.....ChrLit('A')
.....While
.....Le
.....Id(i)
.....ChrLit('Z')
.....StatList
.....Call
.....Id(putchar)
.....Id(i)
.....Store
.....Id(i)
.....Add
.....Id(i)
.....IntLit(1)
.....Return
.....IntLit(0)
```

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 correcta, 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 trabalho deverá ser validado no MOOSHAK, usando o concurso criado especificamente para o efeito em <https://mooshak2.dei.uc.pt/~comp2018/>. Será tida em conta apenas a 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 abstracta, de acordo com a estratégia de desenvolvimento proposta. Note que MOOSHAK não deve ser utilizado como ferramenta de depuração.

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