Projeto realizado no âmbito da cadeira de Compiladores

Compilador para a linguagem Uc

Coimbra, 20 de Dezembro de 2020

João Marcelino PL6 2018279700 jmarcelino@student.dei.uc.pt

Sofia Silva PL6 2018293871 sofiasilva@student.dei.uc.pt

I Gramática re-escrita:

De forma a evitar ambiguidade, fizemos algumas pequenas alterações à gramática fornecida no enunciado.

Estas alterações na gramática foram feitas também com o intuito de facilitar a criação da árvore de sintaxe abstrata (AST).

No ficheiro yacc temos as declarações dos tokens que criámos no ficheiro lex e que serão aceites pelo analisador lexical. A declaração dos tokens foi feita da seguinte forma:

```
%token CHAR
```

Declarámos ainda os símbolos não-terminais aceites pelo analisador lexical,

```
%type <node> FunctionsAndDec.
```

Foi criado novo símbolo FunctionsAndDec de modo a facilitar a criação do nó *Program* na árvore.

Símbolos como:

- FuctionsAndDecExtra
- DeclarationsAndStatementsRep
- DeclarationExtra
- ParameterExtra
- StatementBrace

foram criados de modo a facilitar a visualização de loops na gramática.

FunctionHelper foi gerado para diferenciar uma *Function Definition* de uma *Function Declaration*.

StatementReturn foi criado para agrupar as opções da gramática após ler o token RETURN.

StatementElse foi criado para agrupar as opções da gramática num If statement.

ExprComma foi adicionado de maneira a separar uma expressão COMMA e uma expressão CALL na criação da AST.

```
%%
FunctionsAndDec: FunctionsAndDeclarations
    ;

FunctionsAndDeclarations: TypeSpec FunctionDeclarator FuctionsAndDecExtra
    | Declaration FuctionsAndDecExtra
    | error SEMI FuctionsAndDecExtra
    ;

FuctionsAndDecExtra: FunctionsAndDeclarations
    |
    ;

TypeSpec: CHAR
    | INT
    | VOID
    | SHORT
    | DOUBLE
    ;
```

```
FunctionDeclarator: ID LPAR ParameterList RPAR FunctionHelper       {n
    ;

ParameterList: TypeSpec ParameterDeclaration ParameterExtra         {j
    ;

ParameterExtra: COMMA TypeSpec ParameterDeclaration ParameterExtra  {
    |                                                               {$
    ;

ParameterDeclaration: ID                                            {$
    |                                                               {$
    ;

FunctionHelper:   LBRACE FunctionBody RBRACE                        {
    | SEMI                                                          {
    ;

FunctionBody: DeclarationsAndStatements                             {$
    |                                                               {$
    ;

DeclarationsAndStatements: StatementList DeclarationsAndStatementsRep
    | Declaration DeclarationsAndStatementsRep                      {j

    ;

DeclarationsAndStatementsRep: DeclarationsAndStatements             {$
    |                                                               {$
    ;
```

```
Declaration:TypeSpec Declarator DeclarationExtra
    | TypeSpec error SEMI
    ;

DeclarationExtra: COMMA Declarator DeclarationExtra
    | SEMI
    ;

Declarator:ID ASSIGN Expr
    | ID
    ;

StatementList: Statement
    | error SEMI
    ;

Statement:  SEMI
    | Expr SEMI
    | LBRACE RBRACE

    | LBRACE StatementBrace RBRACE
    | LBRACE error RBRACE

    | IF LPAR Expr RPAR StatementList StatementElse

    | WHILE LPAR Expr RPAR StatementList

    | RETURN StatementReturn

    ;
```

```
StatementBrace: StatementBrace StatementList
    | StatementList
    ;


StatementElse: ELSE StatementList
    | %prec "then"

    ;

StatementReturn: SEMI
    | Expr SEMI
    ;
```

```
Expr: Expr ASSIGN Expr
    | Expr COMMA Expr

    | Expr PLUS Expr
    | Expr MINUS Expr
    | Expr MUL Expr
    | Expr DIV Expr
    | Expr MOD Expr

    | Expr OR Expr
    | Expr AND Expr
    | Expr BITWISEAND Expr
    | Expr BITWISEOR Expr
    | Expr BITWISEXOR Expr

    | Expr EQ Expr
    | Expr NE Expr
    | Expr LE Expr
    | Expr GE Expr
    | Expr LT Expr
    | Expr GT Expr

    | PLUS Expr %prec NOT
    | MINUS Expr %prec NOT
    | NOT Expr


    | ID LPAR ExprComma RPAR
    | ID LPAR RPAR
    | ID LPAR error RPAR
    | LPAR Expr RPAR
    | LPAR error RPAR

    | REALLIT
    | CHRLIT
    | INTLIT
    | ID
    ;

ExprComma: ExprComma COMMA Expr
    | Expr %prec "then"
    ;
```

Em relação à associatividade e precedência de operadores, utilizámos as *keywords* *%left* e *%right* de modo a identificar os *tokens* associativos à esquerda ou à direita, respetivamente. Esta parte tornou-se particularmente relevante de modo a evitar conflitos *shift-reduce* durante a análise sintática. Quanto mais abaixo na lista, maior será a sua precedência, os *tokens* que se encontram na mesma linha, possuem o mesmo grau de precedência.

```
%left COMMA
%right ASSIGN
%left OR
%left AND
%left BITWISEOR
%left BITWISEXOR
%left BITWISEAND
%left EQ NE
%left LE GE LT GT
%left PLUS MINUS
%left MUL DIV MOD
%right NOT
%left LPAR

%right "then" ELSE
```

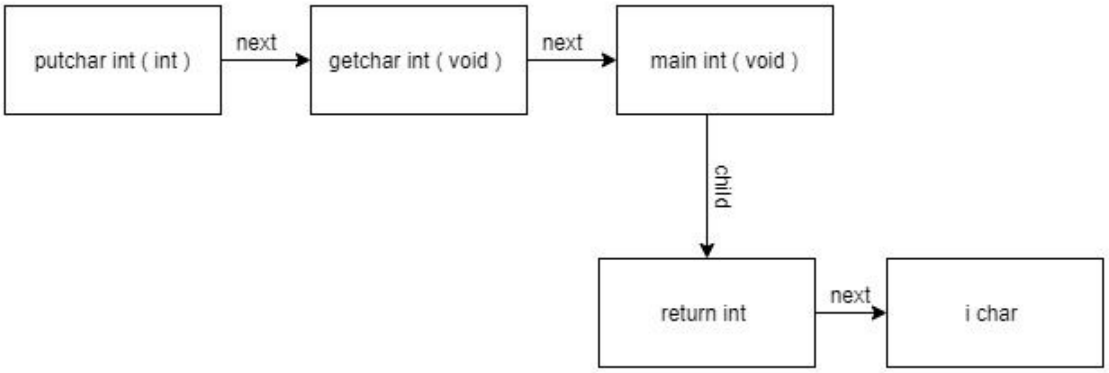
II Algoritmos e estruturas de dados da AST e da tabela de símbolos

De modo a poder construir a árvore de sintaxe abstrata (AST), foi necessária a implementação de código em C. Construímos uma lista ligada, onde cada nó representa um elemento da árvore, esse nó poderá ter filhos e irmãos conforme a estrutura representada no enunciado.

```
typedef struct  node *nodeptr;

typedef struct node{

    char *id;

    char *type;

    nodeptr nodeNext;

    nodeptr nodeBrother;

}Node;
```

- O *id* guarda o valor do token no caso do tipo deste ser Id, ChrLit, IntLit ou RealLit;
- O *type* guarda o tipo do token;
- *NodeNext* aponta para o nó filho;
- *NodeBrother* aponta para o nó irmão.

Em relação à tabela de símbolos, construímos também uma lista ligada onde cada nó representa uma tabela. Esse nó possui também um ponteiro para nó filho que que representa outra lista ligada(com a mesma estrutura) mas em que cada nó representa uma linha da tabela de símbolos da função específica, segue exemplo:




```
typedef struct nodeTable * tableNode;

typedef struct nodeTable{

    char *name;

    char *type;

    paramNode paramList;

    tableNode next;

    tableNode child;

}nodet;
```

- name representa o nome da função, ou o nome da variável declarada;
- type representa o tipo da função ou variável;
- paramList é um ponteiro para a lista de parâmetros;
- next é um ponteiro para o próximo nó na tabela;
- child é um ponteiro que apenas é usado em funções que necessitam de guardar os elementos para a sua tabela.

```
typedef struct nodeParam *paramNode;

typedef struct nodeParam{



    char *name;

    char *var;

    paramNode next;


}nodep;
```

- name, guarda o tipo do parâmetro;
- var, guarda o nome do parâmetro;
- next, aponta para o parâmetro seguinte.

Comentário

O nosso compilador obteve pontuação máxima no Mooshak nas duas primeiras metas. Em relação à terceira meta, tentámos implementar o máximo que conseguimos, no entanto não conseguimos implementar os erros. Como a meta 3 demorou mais tempo do que o previsto, não começámos a meta 4.