

Compiladores

Leandro Pais - uc2017251509

Dezembro 2020

1 Introdução

No âmbito da disciplina de Compiladores (edição 2020) no seguimento do desenvolvimento de um compilador para a linguagem *UC* (um subconjunto da linguagem C) apresenta-se este relatório com o objetivo de esclarecer as opções técnicas tomadas nas diferentes fases do projeto. Fases estas que dizem respeito à gramática, algoritmos, estruturas de dados e por fim a geração de código. No que toca a esta última fase não foram tomadas quaisquer opções pois não foi realizada.

2 Gramática

A gramática fornecida para a linguagem *UC* no enunciado estava em formato *EBNF* que não é reconhecido pelo *yacc* mais especificamente no que toca à utilização de parêntesis retos que representam opcional e os curvos que representam zero ou mais repetições. Assim sendo, foi pensada uma maneira modular e universal para resolver todos estes casos particulares de forma estruturada e que não gerasse confusão. Esta técnica consiste de em ambos os casos criar uma produção extra que no caso dos parêntesis retos apenas contém a produção original e uma produção vazia simulando assim *opcional*. Já no caso dos parêntesis curvos a ideia é semelhante no sentido em que é criada uma produção extra com uma produção vazia mas à produção original é adicionada uma recursão à esquerda simulando assim *zero ou mais vezes*.

Posto isto, tendo já uma gramática capaz de ser reconhecida pelo *yacc*, foi então altura de lidar com todos os erros que foram gerados nomeadamente erros de *shift-reduce (sr)* e *reduce-reduce (rr)*. Para tal, foram adicionadas regras de precedência, de acordo com a seguinte tabela.

C Operator Precedence		
Precedence	Operator	Associativity
1	ELSE	Right-to-Left
2	ASSIGN	Right-to-Left
3	OR	Left-to-Right
4	AND	Left-to-Right
5	BITWISEOR	Left-to-Right
6	BITWISEXOR	Left-to-Right
7	BITWISEAND	Left-to-Right
8	EQ NE	Left-to-Right
9	LT LE GT GE	Left-to-Right
10	PLUS MINUS	Left-to-Right
11	MUL DIV MOD	Left-to-Right
12	LPAR RPAR	Left-to-Right
13	NOT UPLUS UMINUS	Right-to-Left

Desta forma, a grande maioria dos erros ficaram resolvidos. No entanto, é importante diferenciar dois casos particulares que não ficaram resolvidos com esta abordagem requerendo assim uma atenção especial.

O primeiro caso é na produção referente às *DeclarationsAndStatements* pois tanto a produção das *Declarations* como a das *Statements* tratam o erro *SEMI* do lado direito assim ao juntar as duas numa produção dá-se origem a um erro de reduce-reduce em que o yacc não consegue decidir que regra utilizar para reduzir o erro isto resolve-se criando uma produção extra neste caso nas *Statements* sem tratamento de erros e fazê-lo separadamente depois numa produção à parte (Figura 1).

<pre> /*----Production 4----*/ DeclarationsAndStatements: StatementWithoutError DeclarationsAndStatements { \$\$ = \$1; insertNeighbour(\$\$, \$2); } Declaration DeclarationsAndStatements { \$\$ = \$1; insertNeighbour(\$\$, \$2); } StatementWithoutError { \$\$ = \$1; } Declaration { \$\$ = \$1; } ; /*----End Production----*/ </pre>	<pre> Statement: StatementWithoutError { \$\$ = \$1; } error SEMI { \$\$ = NULL; } ; /*----End Production----*/ </pre>
---	--

Figure 1: Declarations and Statements

O segundo caso diz respeito às Comma Expression que devido ao facto de na produção *ID : LPAR [Expr { Comma Expr}] RPAR* ser possível o uso de *Comma Expression* bem como na *Expression* em si, através da produção *Expr: Expr COMMA Expr*. Novamente isto dá origem a um erro *rr*, assim sendo a solução passa mais uma vez por criar uma produção extra onde é separado a produção *Expr: Expr COMMA Expr* das restantes expressões para ser tratada depois numa produção à parte (Figura 2).

```

COMMAExpr: COMMAExpr COMMA Expr
{
    $$ = buildNode("Comma", "", 1, 0, nline, ncol);
    insertChild($$, $1);
    insertChild($$, $3);
}
| Expr
{
    $$ = $1;
}
;

/*Auxiliary production to handle zero or more <COMMA Expr>*/
XPRBRACE: COMMA Expr XPRBRACE
{
    $$ = $2;
    insertNeighbour($$, $3);
}
|
{
    $$ = NULL;
}
;
/*----End Production----*/

```

Figure 2: Expression Comma Expression

Assim, dá-se por concluída a fase do tratamento da gramática com todos os erros e conflitos devidamente resolvidos.

3 Algoritmos e Estruturas de Dados

Começando pelas estruturas, é importante olhar para dois aspetos importantes deste projeto.

Em primeiro lugar, na análise sintática, temos a árvore de sintaxe abstrata (Abstract Syntax Tree - AST) em que cada nó contém os campos com toda a informação necessária referente ao token e também os campos necessários para a construção da árvore, nomeadamente um ponteiro para um nó filho e um outro ponteiro para uma lista ligada de nós vizinhos como se pode ver na figura 3.

```

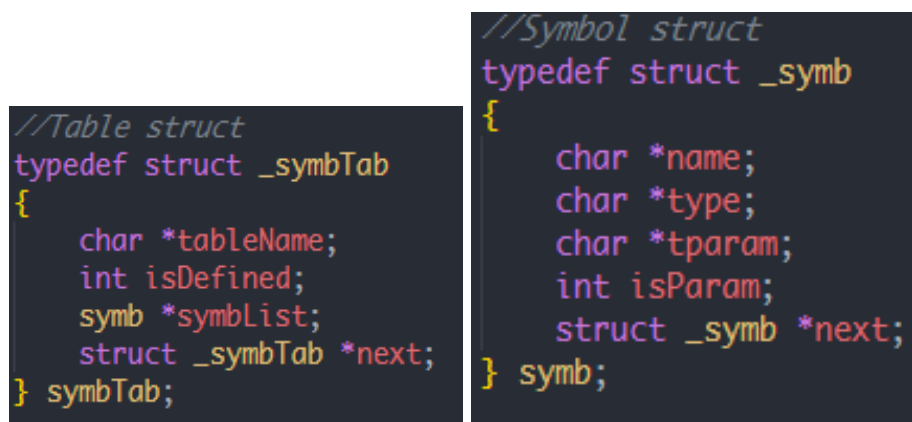
typedef struct ASTNode
{
    /* Node data*/
    //Token type are the ones explicit such as Program and others
    char *tokenType;
    //Tokens such as ID can have a lexical value
    //ASTNode needs to be able to store it
    char *tokenVal;
    char *note;
    int isExpression;
    int isOp;
    int line;
    int col;
    /* Node structs to build the tree*/
    //Children (one level deeper)
    struct ASTNode *child;
    //Neighbour (same level)
    //Linked list
    struct ASTNode *next;
} ASTNode;

```

Figure 3: AST - Data Structures

Como se pode ver, esta abordagem difere daquela que foi proposta nas aulas laboratoriais que previa a criação de uma estrutura para cada tipo de token, no entanto visto que nos foi dada liberdade para tal, foi decidido utilizar uma abordagem mais modular utilizando um string para o tipo de token e uma outra string para o seu valor, esta última está vazia quando o tipo de token não requer a existência dum valor, ou seja, não é nem um *ID*, *IntLit*, *RealLit* ou *ChrLit*. Os restantes campos foram adicionados conforme houve necessidade para tal, são exemplo disto o campo *note*, que serve para guardar a anotação feita na árvore, os campos *line*, *col* para indicar erros com precisão, entre outros. A esta

estrutura estão associadas as respectivas funções de alocação, criação, edição e respectiva libertação de memória já inerentes a estas estruturas de dados que são as árvores que foram adaptadas para servir os propósitos do projeto. É de realçar que, como foi referido, esta estrutura foi sem dúvida a que mais alterações sofreu ao longo do projeto assim sendo o design modular destas funções foi essencial para encurtar algumas das tarefas mais dispendiosas em termos de tempo. Por último é importante ainda olhar para a análise semântica, onde temos uma estrutura mais simples, uma lista ligada que mantém informação sobre a declaração de variáveis/funções, os seus tipos e no caso das funções o valor de retorno (Ver Figura 4).



```
//Table struct
typedef struct _symbTab
{
    char *tableName;
    int isDefined;
    symb *symbList;
    struct _symbTab *next;
} symbTab;

//Symbol struct
typedef struct _symb
{
    char *name;
    char *type;
    char *tparam;
    int isParam;
    struct _symb *next;
} symb;
```

Figure 4: Linked Lists - Data Structures

Esta estrutura permite, como foi dito, realizar a análise semântica capaz de escrutinar um programa com um maior nível de detalhe no que toca a erros e warnings tendo em conta as várias scopes do mesmo. A estrutura em si, como já foi referido, é relativamente simples não passa de uma lista ligada de tabelas com a informação referente a cada função definida (neste caso é apenas o nome), é importante saber se está definida ou apenas declarada daí o parâmetro *isDefined*, e um ponteiro para a seguinte, é de notar que o primeiro elemento desta lista é sempre a tabela *Global* do programa. A estrutura tem ainda um ponteiro para uma outra lista ligada de símbolos que dizem respeito às variáveis de cada função, como já foi mencionado. A esta estrutura estão associadas as respetivas funções de alocação, criação, edição e respectiva libertação de memória que nada têm de especial por se tratar de uma estrutura simples.

4 Geração de Código

Como referido na introdução esta secção encontra-se ausente pois esta fase do projeto não chegou a ser desenvolvida.

5 Conclusão

Para concluir e em retrospectiva, as opções tomadas ao longo do projeto deveriam ter sido melhor pensadas numa fase anterior à implementação, no sentido de as tornar mais modulares, pois foram sofrendo várias alterações no decorrer do projeto. Alterações essas que por serem muito dispendiosas em termos de tempo impactaram negativamente a minha performance no trabalho mais especificamente no que toca à quarta meta. No entanto, no geral o projeto foi uma experiência positiva não só no que toca à experiência e à aprendizagem de conceitos sobre compiladores mas também na programação como um todo.

References

- [1] C99 Official Language Manual
https://eden.dei.uc.pt/~rbarbosa/C99_standard.pdf.