Relatório do Projeto de Compiladores

Compilador Juc

Estudante	João Moreira	2020230563
Estudante	Tomás Pinto	2020224069
Professor	Alexandre Jesus	

1 Gramática da Linguagem

A gramática da linguagem foi fornecida no enunciado do projeto em notação EBNF, pelo que sentimos a necessidade de re-escrever a gramática de forma a eliminar as suas ambiguidades, que resultam em conflitos *"Shift-Reduce"* e *"Reduce-Reduce"*. Desta forma, a ferramenta *yacc* pode fazer a análise sintática ascendente sem problemas.

De modo a eliminar as ambiguidades, introduzimos regras que determinaram a associatividade e precedência de uns operadores em relação aos outros. Isto foi feito atravez dos comandos *"%left"*, *"%right"* e *"%nonassoc"*. A declaração de associatividade tem maior precedência de baixo para cima.

Na seguinte tabela está presente a nossa tomada de decisão:

Operador	Associatividade
ASSING	right
OR	left
AND	left
XOR	left
EQ, NE	left
LT, GT, LE, GE	left
LSHIFT RSHIFT	left
PLUS MINUS	left
STAR DIV MOD	left
NOT	right
LPAR, RPAR, LSQ RSQ	left

Segundo a documentaçao da linguagem Juc, nas instruções *if-then-else*, o *else* pode ser omitido. Sendo assim, foi criado o operador auxiliar não associativo *NO_ELSE*. Neste caso recorremos ao comando *"%prec"*, cujo altera o nível de precedencia associado com uma regra da gramática. Assim, podemos dar maior prioridade ao *else* e a mesma na sua ausência.

Operador	Associatividade
NO_ELSE	nonassoc
ELSE	left

Outras modificações que aplicamos na gramática estão relacionadas à implementação de ciclos que permitam produzir multiplas declarações e definições. Para estas modificações implementamos produções recursivas à direita. Tomamos esta decisão motivada pela forma que estavamos a imprimir a AST. Um exemplo pode ser na produção *"Statement"*.

```
Statement: LBRACE Statement2 RBRACE

Statement2: Statement Statement2
          |
          ;
```

2 Estruturas de Dados

2.1 Token

Para que fosse possivel guardar a informação dos tokens lidos pelo lex, criamos o tipo *token_t*, cujo é uma estrutura que guarda o valor e posição respetiva ao token.

```
typedef struct Token {
    char *value;
    int line;
    int col;
} token_t;
```

2.2 AST

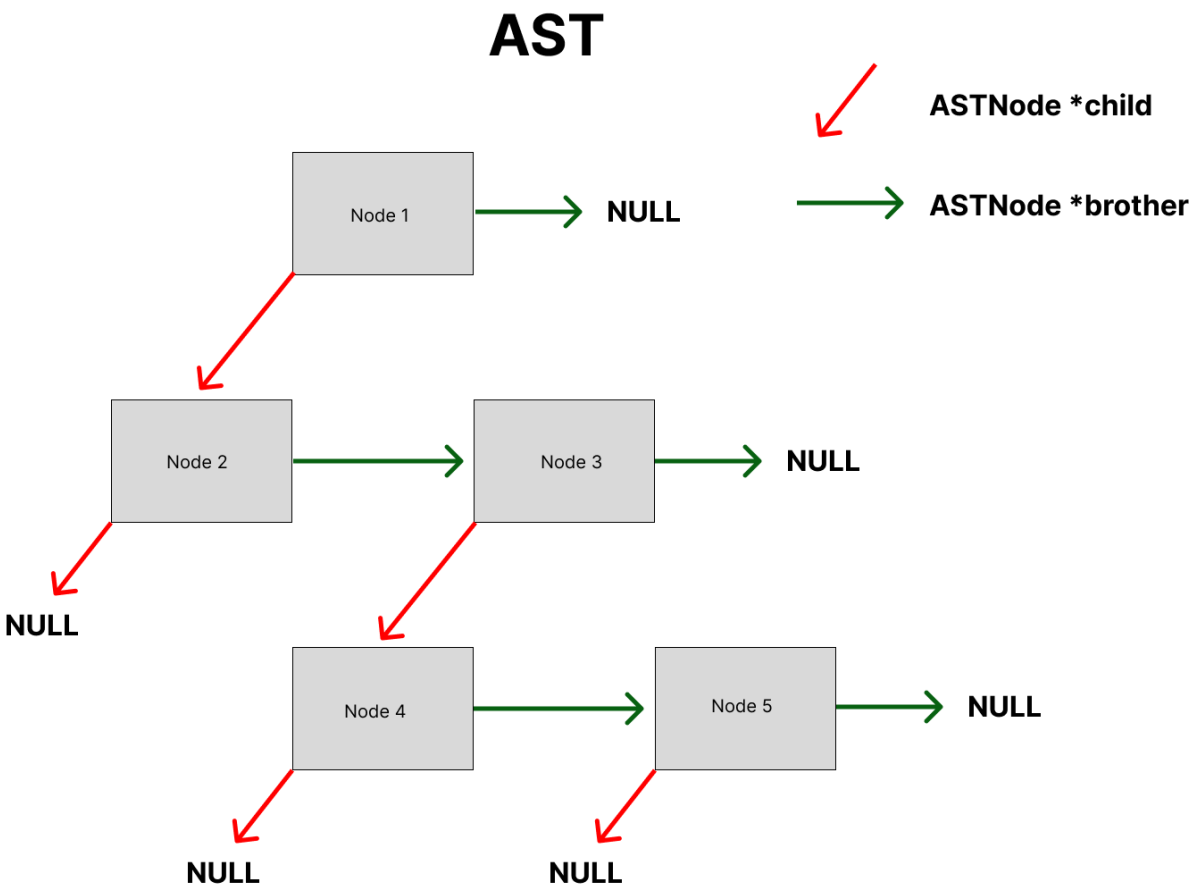
Para implementar a arvore de sintaxe abstrata (AST) começamos por criar uma estrutura para representação dos nós da arvore.

Apresenta-se de seguida o modelo de cada nó:

```
typedef struct _ASTNode {
    char *id;
    char *value;
    char *type;
    int line;
    int col;

    struct _ASTNode *child;
    struct _ASTNode *brother;
} ast_node_t;
```

- char *id: Nome do nó ("MethodDecl", "FieldDecl", ...)
- char *value: Valor do token (yytext da analise lexical)
- char *type: Tipo do nó (anotado na análise semântica)
- int line: Linha onde o token foi identificado
- int col: Coluna onde o token foi identificado
- struct _ASTNode *child: Ponteiro para o primeiro nó filhe de um certo nó.
- struct _ASTNode *brother: Ponteiro para o irmão de um certo nó (proximo filho de um nó pai).



[!]

2.3 Tabela de Símbolos

Para a criação da tabela de símbolos criamos uma estrutura auxiliar *param_list*. Esta permite-nos criar uma lista ligada de parametros.

```
typedef struct _param_list {
  char *param;

  struct _param_list *next;
} param_list;
```

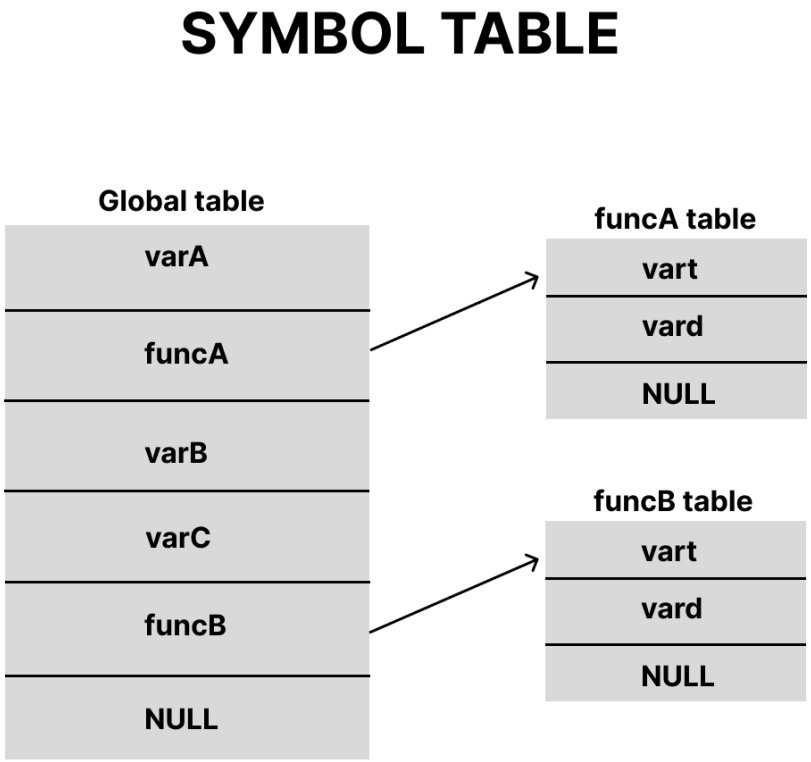
A estrutura principal é *symbol_table* que permite-nos guardar toda a informação relativa a um símbolo da tabela. O que fizemos foi criar uma lista de símbulos global, onde cada símbolo, no caso de corresponder a um método (declaração de função) aponta para outra lista de símbolos.

```
typedef struct _symbol_table {
    char *id;
    char *value;
    bool is_param;
    bool is_func;
    param_list *params;

    struct _symbol_table *symbols;
    struct _symbol_table *next;
} symbol_table;
```

- param_list:
  - char *param: tipo de parametro (int, double, boolean...)
  - struct_param_list *next: ponteiro para o próximo parametro
- symbol_table:

- char *id: id obtido do lex
- char *value: valor do id
- bool is_param: true no caso o símbolo pertencer a um parametro
- bool is_func: true no caso o símbolo pertencer a uma função
- param_list *params: ponteiro para a lista de parametros
- struct_symbol_table *symbols: ponteiro para o próximo símbolo
- struct_symbol_table *next: ponteiro para a próxima lista de símbolos



2.4 Algoritmos

Os algoritmos utilizados foram essencialmente algoritmos de travessia de listas e árvores.

A pesquisa na árvore é realizada de forma recursiva. Inicialmente percorremos os irmãos, o que nos permite ter acesso imediato às variávies e funções globais. Estas, serão adicionadas no final da tabela global.

No caso de estarmos numa função, consultamos primeiro os filhos e depois os irmãos. Na fase ascendente da recursão adicionamos da mesma forma à respetiva tabela e anotamos a ast.