1 2  9 0 UNIVERSIDADE D COIMBRA

Projeto Compiladores

Compilador para a linguagem deiGo

Trabalho realizado por:

- Pedro Afonso Ferreira Lopes Martins n.º 2019216826
- João Carlos Borges Silva n.º 2019216753

Índice:

1. Gramática reescrita:

Primeiramente procedemos à identificação dos tokens como constava no enunciado, de seguida começámos a estabelecer as expressões regulares e os estados necessários para fazer uma análise lexical correta e tratar deste tipo de erros, caso eles surjam.

Passando para o yacc, numa fase inicial começámos por transcrever em conjunto a gramática em notação EBNF a nós fornecida para o nosso programa. Ao realizar esta etapa deparámo-nos com os conflitos de "shift-reduce" e "reduce-reduce". Como tal, removemos as ambiguidades da nossa gramática, através da introdução de regras que estabelecem a associatividade e precedência dos operadores entre si, recorrendo aos comandos %left, %right e %nonassoc para este efeito. De seguida, apresentamos a lista de comandos por nós usada, ordenada por ordem crescente de precedência.

- %left COMMA
- %right ASSIGN
- %left OR
- %left AND
- %left EQ NE GT GE LT LE
- %left MINUS PLUS
- %left STAR DIV MOD
- %right NOT

E de seguida os %nonassoc:

- %nonassoc NO_ELSE
- %nonassoc ELSE
- %nonassoc RPAR LPAR LSQ RSQ

Sendo que o primeiro parâmetro representa a associatividade do operador.

É de notar que nos operadores com associatividade %nonassoc, estes são usados na gramática com recurso ao %prec de forma atribuir à regra aplicada uma precedência de uma variável. Isto indica-nos que os operadores "(", "[", "]", ")" são os que apresentam maior prioridade, sendo precedidos da regra if-else e, de seguida, else.

De seguida aplicámos algumas alterações à gramática fornecida, de forma a incluir as situações de erro, entre as quais:

- | ID LPAR error RPAR
- | LPAR error RPAR
- | ID COMMA BLANKID ASSIGN PARSEINT LPAR error RPAR
- | error

2. Algoritmos e estruturas de dados da AST e da tabela de símbolos:

Ao chegar a esta etapa tivemos primeiro de definir as estruturas a utilizar na nossa árvore de sintaxe.

```
typedef struct ast_node
ast_tree;
struct ast_node{
     token* token;
     char* value;
     ast_tree* first_child;
     ast_tree* next_sibling;
};
```

```
typedef struct token_ token;
struct token_{
     char* id;
     char* type;
     int line;
     int column;
     int global;
     int is_defined;
     int is_func;
     int used;
};
```

Estruturas referentes aos nós e tokens.

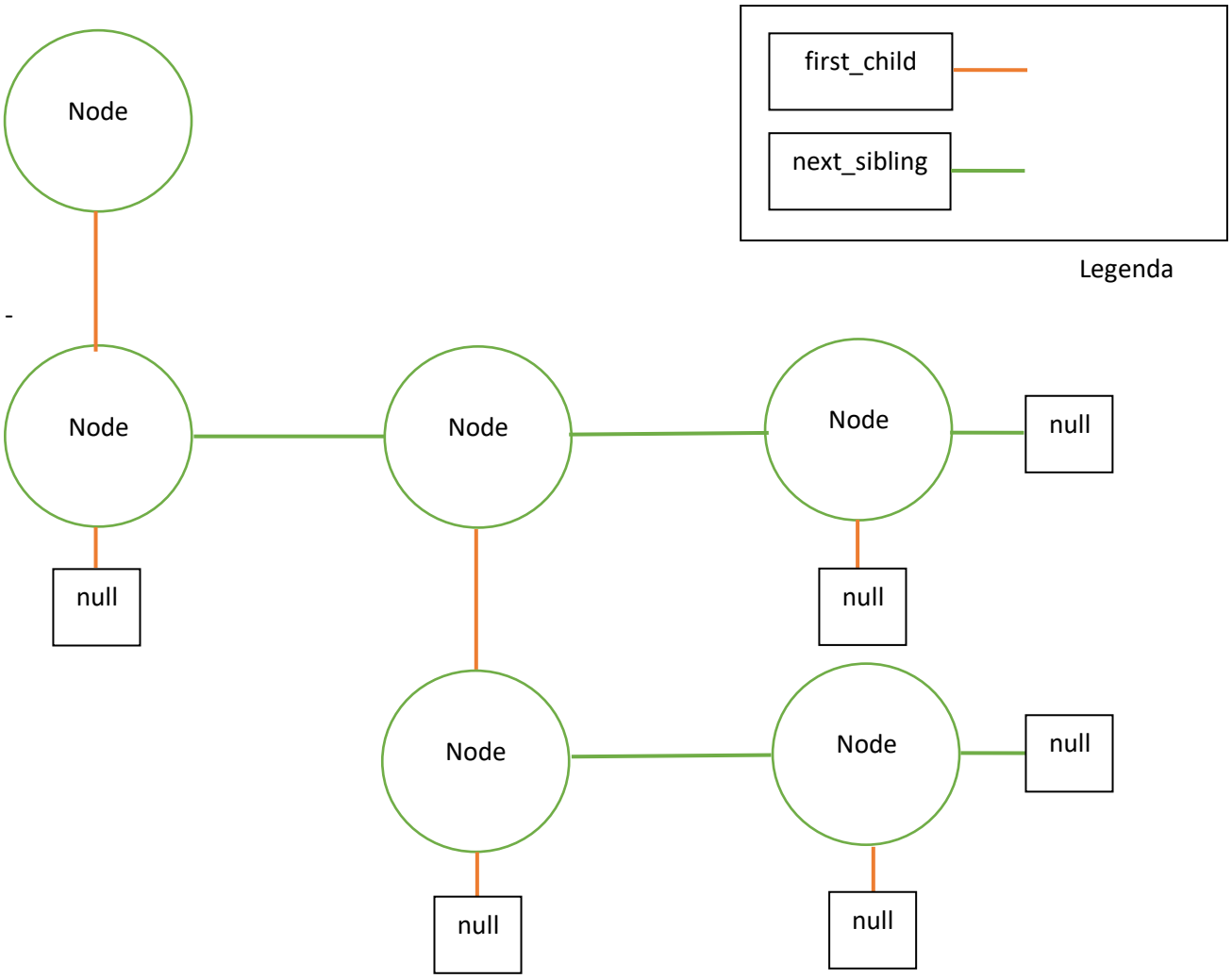
Relativamente à estrutura **token**:

- **Id**: Corresponde ao nome do nó do token.
- **Type**: Corresponde ao tipo (int, float32, bool, string) do token
- **Line**: A linha onde o token foi identificado.
- **Column**: A coluna onde o token foi identificado.
- **Global**: Indica de o token é local ou global.
- **Is_defined**: Indica se está definido no programa (Usado para erros)
- **Is_func**: Indica se o token é uma função.
- **Used**: Indica se o token foi ou não usado.

Relativamente à estrutura **ast_tree**:

- **Token**: A estrutura acima mencionada e explicada.
- **Value**: O valor (yytext) de um nó, caso ele tenha.
- **First_child**: Ponteiro para o nó que corresponde ao primeiro filho.
- **Next_sibling**: Ponteiro para o nó que corresponde ao próximo nó (irmão).

De seguida procedemos a pensar no algoritmo a implementar nesta etapa e como é que o iriamos realizar. Como tal, segue-se uma demonstração de como funciona a AST:



Algoritmo da AST

Com a estrutura implementada e o algoritmo pensado, começámos a desenvolver as funções necessárias para efetuar a inserção na árvore. Primeiramente programámos a função **ast_node** que cria um novo nó com os parâmetros fornecidos no yacc (recorrendo à função **new_token** como função auxiliar, de forma a poder criar um token para o nó a adicionar). De seguida procedemos a criar as funções **add_childs** e **add_siblings**, que adicionam um nó como filho e irmão de outro, respetivamente. Por fim, criámos as funções **print_tree** (que imprime a AST inteira por ordem) e a **free_tree** (responsável por libertar a memória previamente alocada na árvore).

De seguida passámos às tabelas de símbolos com o intuito de armazenar dados sobre as variáveis globais e locais do programa e as funções definidas:

```
typedef struct table_ table;
struct table_{
    char* name;
    char* nametoprint;
    char* type;
    char* params;
    symbol* first_symbol;
    table* next_table;
};
```

```
typedef struct symbol_ symbol;
struct symbol_{
    char* id;
    char* type;
    char* whatisit;
    char* params;
    int line;
    int column;
    bool used;
    bool is_defined;
    bool is_func;
    int register_tab;
    symbol* next_symbol;
};
```

Estruturas referentes aos nós e tokens.

Relativamente à estrutura **table**:

- **Name**: Nome da função (ou tabela global).
- **Nametoprint**: Nome da função a imprimir no topo da tabela.
- **Type**: Tipo da função.
- **Params**: Parâmetros da função.
- **First_symbol**: Ponteiro para o primeiro símbolo da tabela tipo **struct symbol.**
- **Next_table**: Ponteiro para a próxima tabela do tipo **struct table.**

Relativamente à estrutura **symbol**:

- **Id**: Nome do símbolo.
- **Type**: Tipo do símbolo.
- **Whatisit**: Indicador sobre se o símbolo é um parâmetro, variável ou função.
- **Params**: Caso seja função, lista os parâmetros da mesma.
- **Line**: Linha fornecida da AST para apresentação de erros semânticos.
- **Column**: Coluna fornecida da AST para apresentação de erros semânticos.
- **Used**: Verifica se um símbolo foi ou não usado para verificação de erros.
- **Is_defined**: Verifica se um símbolo foi definido para verificação de erros.
- **Is_func**: Confirma se o símbolo é uma função ou não.
- **Next_symbol**: Ponteiro para o próximo símbolo de uma tabela.
- **register_tab:** Indicador do registo do símbolo para o LLVM.

Em termos de tabela de símbolos, começamos por uma **struct table** global que armazena todas as declarações de variáveis globais e funções. À medida que uma função é encontrada, é criada uma nova tabela, local, que contém informações relativas às variáveis locais e seus parâmetros e tipo da função. Esta representa uma **struct table** e é associada à tabela anterior através do parâmetro "**next_table**" da **struct table**. Parâmetros, variáveis e funções são adicionados como símbolos, sendo o primeiro de uma função ou global representado pelo "**first_symbol**" da struct table e símbolos sucessivos representados por "**next_symbol**" na **struct symbol**.

3. Geração de código LLVM:

Nesta etapa final iremos gerar o código LLVM para o input apresentado. Para isso iremos recorrer à função **generateLLVM** que irá receber o primeiro nó da AST gerada e a tabela global de símbolos. De seguida irá tratar do caso vazio (ficheiro de entrada vazio) e dos dois filhos que pode ter (declaração de variável e declaração de função) sendo a mesma uma função que corre do início ao fim da análise da AST, terminando quando a árvore inteira for percorrida.

No caso de ser uma variável recorremos à função **varDeclLLVM**. Esta distingue variáveis globais e locais e, com o auxílio da função **typeLLVM** (devolve o tipo LLVM de um nó), fornece o código LLVM para efetuar a declaração.

No caso de ser função, iremos tratar primeiro do header da mesma e os seus parâmetros, acautelando o caso em que a função tem o nome de main. Usando como auxiliar as funções **typeLLVM** e **getFuncParams** (dá print dos parâmetros de uma função no formato LLVM) temos todas as ferramentas necessárias para concluir esta etapa inicial.

De seguida prosseguimos para o corpo da função, podendo o mesmo gerar várias situações, listadas abaixo:

- **Call:** Função **callLLVM** recursiva que recorre à função auxiliar **loadLLVM** para dar load dos parâmetros da função. Capaz de trabalhar com parâmetros do tipo **call**, **id** e **literal**.
- **Print:** Função **printLLVM** responsável por tratar dos prints da função. Usa typeLLVM como auxiliar para definir o tipo LLVM do dado a imprimir e, de seguida, verifica se representa um operador ou id/dado não declarado.
- **If:** Não implementado no nosso programa.
- **Return:** Responsável por controlar o nó **return** em LLVM através da função **returnLLVM**. Capaz de tratar de **id**, **literal** e ainda ter **calls**, o que é referido à função **callLLVM** acima referida.
- **VarDecl:** Função varDeclLLVM que recebe por parâmetro que a variável é local e recorre à função auxiliar **allocaLLVM** para escrever o código LLVM responsável por alocar o espaço destinado à variável, usando para esse efeito o contador local da função.
- **For:** Não implementado no nosso programa.
- **Assign:** Função **assignLLVM** responsável por atribuir a um **id** um valor. Caso este valor seja um **id** recorre-se à função **loadLLVM** para dar load do mesmo.

Por fim recorre-se à função **storeLLVM** para escrever em LLVM o código correspondente ao armazenamento dos dados, fornecendo à mesma o contador da função e o contador do id.

- **Operador:** Função **operationLLVM** responsável por tratar dos operadores presentes na função. Função recursiva até encontrar todos os operadores abaixo de onde foi invocado, enquanto devolve o código LLVM do operador e o load do resultado obtido.