UNIVERSIDADE D
COIMBRA

Universidade de Coimbra
Faculdade de Ciências e Tecnologias
Departamento de Engenharia Informática
Licenciatura de Engenharia Informática

Compiladores

Relatório da Entrega Final

João Afonso dos Santos Simões
2022236316
Rodrigo Miguel Santos Rodrigues
2022233032

Coimbra,
15 de dezembro de 2024

Índice

1. Gramática Re-escrita

1.1 Declarations

Declarations: -

{VarDeclaration SEMICOLON | FuncDeclaration SEMICOLON}

```
Declarations:
    VarDeclaration SEMICOLON Declarations
    | FuncDeclaration SEMICOLON Declarations
    | /*  null production */
```

O símbolo **declarations** pode gerar uma produção VarDeclaration ou uma produção FuncDeclaration, tal como pode gerar uma produção vazia. A geração de ambas é realizada pela recursividade à esquerda.

1.2 VarSpec

VarSpec: -

IDENTIFIER {COMMA IDENTIFIER} Type

```
VarSpec:
    IDENTIFIER VarSpecAux Type
```

```
VarSpecAux:
    COMMA IDENTIFIER VarSpecAux
    | /*  null production */
```

Para lidar com várias declarações de variáveis do mesmo tipo, foi criado um símbolo auxiliar, que permite a produção de vários identifiers antecedidos por uma vírgula com a recursividade à direita. A lista de identifiers(pode ser só 1) termina com a produção nula.

1.3 FuncDeclaration

A produção dada no enunciado

FuncDeclaration : FUNC IDENTIFIER LPAR [Parameters] RPAR [Type] FuncBody

deu origem a

FuncDeclaration:
FUNC IDENTIFIER LPAR Parameters RPAR Type FuncBody
FUNC IDENTIFIER LPAR RPAR FuncBody
FUNC IDENTIFIER LPAR Parameters RPAR FuncBody
FUNC IDENTIFIER LPAR RPAR Type FuncBody

respeitando as combinações existentes de Parameters(P) e Type(T): PT, _ , P, T.

1.4 Parameters

Neste símbolo, a ideia é a mesma que foi usada em VarSpec.

Parameters: -
IDENTIFIER Type {COMMA IDENTIFIER Type}

```
Parameters:
    IDENTIFIER Type ParametersAux
```

```
ParametersAux:
    COMMA IDENTIFIER Type ParametersAux
    | /*  null production */
```

A criação do símbolo ParametersAux permite a declaração de vários parâmetros com a recursividade à direita. A lista de parâmetros (pode ser só um) termina com a produção nula.

1.5 VarsAndStatements

Neste símbolo repete-se o que foi feito em FuncDeclaration.

VarsAndStatements. -
VarsAndStatements [VarDeclaration | Statement] SEMICOLON | ϵ

```
VarsAndStatements:
    VarsAndStatements VarDeclaration SEMICOLON
    | VarsAndStatements Statement SEMICOLON
    | VarsAndStatements SEMICOLON
    | /* null production */
    ;
```

A recursividade à esquerda é responsável pela produção de várias variáveis e statements.

1.6 Statement

Statement ⟶ IDENTIFIER ASSIGN Expr
Statement ⟶ LBRACE {Statement SEMICOLON} RBRACE
Statement ⟶ IF Expr LBRACE {Statement SEMICOLON} RBRACE [ELSE LBRACE {Statement SEMICOLON} RBRACE] -
Statement ⟶ FOR [Expr] LBRACE {Statement SEMICOLON} RBRACE
Statement ⟶ RETURN [Expr]
Statement ⟶ FuncInvocation | ParseArgs
Statement ⟶ PRINT LPAR (Expr | STRLIT) RPAR

Statement:
IDENTIFIER ASSIGN Expr
| LBRACE StatementAux RBRACE
| IF Expr LBRACE StatementAux RBRACE
ELSE LBRACE StatementAux RBRACE
| IF Expr LBRACE StatementAux RBRACE
%prec ELSE_IF
| FOR Expr LBRACE StatementAux RBRACE
| FOR LBRACE StatementAux RBRACE
| RETURN Expr
| RETURN
| FuncInvocation
| ParseArgs
| PRINT LPAR Expr RPAR
| PRINT LPAR STRLIT RPAR

```
StatementAux:
    Statement SEMICOLON StatementAux
    | /*  null production */
```

Em statements foram usadas as ideias já previamente definidas.

1.7 FuncInvocation

Neste símbolo foi criado um símbolo auxiliar assim como uma nova produção para as invocações de funções que não recebam nenhum argumento.

FuncInvocation: -
IDENTIFIER LPAR [Expr {COMMA Expr}] RPAR

```
FuncInvocation:
    IDENTIFIER LPAR Expr FuncInvocationAux RPAR
    | IDENTIFIER LPAR RPAR

FuncInvocationAux:
    COMMA Expr FuncInvocationAux
    | /*  null production */
```

1.8 Nota

Nos símbolos Statement, ParseArgs, FuncInvocation e Expression foi adicionada uma produção de erro.

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

Para a criação da AST usámos as seguintes estruturas:

```
struct node {
    struct token* token;
    struct node* child;
    struct node* brother;
};
```

```
struct token {
    char* value;
    category category;
    type type;
    char* annotation;
    int line;
    int column;
};
```

A struct **node** representa um nó na árvore de sintaxe abstrata. É composta por duas structs node que correspondem ao primeiro filho e ao primeiro irmão respetivamente. A struct token guarda o valor do nó (se tiver), categoria, tipo (mais usado para a anotação) e a linha e a coluna onde o token apareceu.

Para a criação das tabelas de símbolos usámos:

```
struct table {
    char* name;
    char* type;
    char* params;
    struct symbol_list* first_symbol;
    struct table* next;
};
```

```
struct symbol_list {
    char* identifier;
    type type;
    int is_param;
    int is_func;
    int is_used;
    int is_declared;
    int is_declared_codegen;
    int is_global;
    struct node* node;
    struct symbol_list* next;
};
```

A struct table representa uma tabela. É composta pelas strings name, type e params que guardam o nome da tabela, tipo e os seus parâmetros. É composta também pela sua lista de símbolos e um ponteiro para a próxima tabela. A lista de tabelas começa sempre pela tabela de símbolos global.

A struct symbol_list é composta pelo identifier do símbolo e tipo, e por um conjunto de flags para o bom funcionamento do compilador. Tem também um ponteiro para o nó associado ao símbolo assim como um ponteiro para o próximo símbolo da lista.

A AST é construída principalmente com as funções new_node(...), add_child(....) e add_brother(...). A anotação da mesma é feita de forma recursiva, começando a anotar os símbolos terminais e de seguida anotar os nós acima.

3. Geração de código

A geração do código começa por definir as funções printf e atoi da linguagem C, assim como as constantes int, float, strlit, true e false. Se existirem strings no código, são também declaradas no início. O mesmo se aplica a variáveis globais.

Dentro das funções, são usados registos temporários para guardar o valor dos parâmetros. Depois é gerado o código do corpo da função. Esta etapa é feita recursivamente, primeiro é gerado o código dos símbolos terminais da árvore, e depois os nós acima.

Nos casos em que o nó é um Assign ou Identifier, é verificado se o token é uma variável local ou global. Em casos de duas variáveis com o mesmo nome, uma global e outra local, é verificado se o identifier a ser gerado é referente à variável global ou local.

Para gerar o código do If são criadas branches para quando a condição for verdadeira, falsa e a branch “end” para o código continuar. No caso do for, são criadas as branches para a condição, corpo e fim. No código do If e dor For, é chamada a função recursiva em cada uma das branches para gerar código.