FCTUC FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE DE COIMBRA

Compiladores

Compilador para a linguagem Juc

António Lopes	PL2	2017262466	uc2017262466@student.uc.pt
Tiago Fernandes	PL2	2017242428	tfernandes@student.dei.uc.pt

Gramática reescrita

Para esta gramática, usamos uma gramática semelhante à do enunciado, com algumas alterações de modo a não causar ambiguidades.

Em caso de existência de ciclos e/ou ambiguidades, criámos símbolos não-terminais (p. e., *MethodInvocationOpt*, *VarDeclOpt*, *Expr1*, *FormalParamsOpt*,…), corrigindo assim esses problemas.

No caso das expressões, criámos 2 estados que representam quase o mesmo (*Expr* e *Expr1*). A separação deve-se ao facto de ter existido uma ambiguidade entre o *Assignment* e *Expr*. No caso do *Expr* é reconhecida uma expressão, sendo que pode ser um *Assignment* ou então uma *Expr1*, onde estão contemplados todos os casos fornecidos no enunciado (Fig. 1).

```
Expr:       Expr1
    |       Assignment

Assignment: ID ASSIGN Expr

Expr1:      LPAR Expr RPAR
    |       Expr1 PLUS Expr1
    |       Expr1 MINUS Expr1
    |       ...
```

Fig. 1: Caso das *Expr*, *Assignment* e *Expr1*

Relativamente às precedências de símbolos, utilizámos os operadores de associatividade *%left* e *%right*, bem como o *%nonassoc* que quanto mais abaixo estiverem na lista das regras (Fig. 2), maior é a sua precedência comparativamente com as outras. Como base para escrever esta lista de regras, utilizámos a documentação do Java.

```
%left ARROW
%left COMMA
%right ASSIGN
%left OR
%left AND
%left XOR
%left NE EQ
%left GE LE LT GT
%left LSHIFT RSHIFT
%left PLUS MINUS
%left STAR DIV MOD
%right NOT
%left RSQ LSQ RPAR LPAR

%nonassoc IF
%nonassoc ELSE
```

Fig. 2: Lista de regras de precedência

```
Program:    CLASS ID LBRACE ProgramOpt RBRACE

ProgramOpt: MethodDecl ProgramOpt
    |       FieldDecl ProgramOpt
    |       SEMICOLON ProgramOpt
    |       /* vazio */

MethodDecl: PUBLIC STATIC MethodHeader MethodBody

FieldDecl:  PUBLIC STATIC Type ID FieldDeclOpt SEMICOLON
    |       error SEMICOLON

FieldDeclOpt:   COMMA ID FieldDeclOpt
    |           /* vazio */

Type:       BOOL
    |       INT
    |       DOUBLE

MethodHeader:   Type ID LPAR MethodHeaderOpt2 RPAR
    |           VOID ID LPAR MethodHeaderOpt2 RPAR


MethodHeaderOpt2:   FormalParams
    |           /* funcao nao tem argumentos */

FormalParams:   Type ID FormalParamsOpt
    |           STRING LSQ RSQ ID

FormalParamsOpt:    COMMA Type ID FormalParamsOpt
    |                /* a funcao so tem um argumento */
```

```
MethodBody:     LBRACE MethodBody2 RBRACE

MethodBody2:    Statement MethodBody2
    |           VarDecl MethodBody2
    |           /* vazio */

VarDecl:    Type ID VarDeclOpt SEMICOLON

VarDeclOpt: COMMA ID VarDeclOpt
    |       /* so tem um parametro definido */

Statement:  LBRACE StatementOpt RBRACE
    |       IF LPAR Expr RPAR Statement
    |       IF LPAR Expr RPAR Statement ELSE Statement
    |       WHILE LPAR Expr RPAR Statement
    |       RETURN StatementOpt3 SEMICOLON
    |       StatementOpt4 SEMICOLON
    |       PRINT LPAR StatementOpt5 RPAR SEMICOLON
    |       error SEMICOLON

StatementOpt:   Statement StatementOpt
    |           /* vazio */

StatementOpt3:  Expr
    |           /* vazio */

StatementOpt4:  MethodInvocation
    |           Assignment
    |           ParseArgs
    |           /* vazio */
```

```
StatementOpt5:  Expr
              | STRLIT

MethodInvocation:  ID LPAR MethodInvocationOpt RPAR
                 | ID LPAR error RPAR

MethodInvocationOpt:  Expr MethodInvocationOpt2
                    | /* vazio */

MethodInvocationOpt2:  COMMA Expr MethodInvocationOpt2
                     | /* vazio */

Assignment: ID ASSIGN Expr

ParseArgs:  PARSEINT LPAR ID LSQ Expr RSQ RPAR
          | PARSEINT LPAR error RPAR

Expr: Expr1
    | Assignment
```

```
Expr1: LPAR Expr RPAR
     | LPAR error RPAR
     | Expr1 PLUS Expr1
     | Expr1 MINUS Expr1
     | Expr1 STAR Expr1
     | Expr1 DIV Expr1
     | Expr1 MOD Expr1
     | Expr1 AND Expr1
     | Expr1 OR  Expr1
     | Expr1 XOR Expr1
     | Expr1 LSHIFT Expr1
     | Expr1 RSHIFT Expr1
     | Expr1 EQ Expr1
     | Expr1 GE Expr1
     | Expr1 GT Expr1
     | Expr1 LE Expr1
     | Expr1 LT Expr1
     | Expr1 NE Expr1
     | NOT Expr1
     | MINUS Expr1 %prec NOT
     | PLUS Expr1 %prec NOT
     | MethodInvocation
     | ParseArgs
     | ID
     | ID DOTLENGTH
     | INTLIT
     | REALLIT
     | BOOLLIT
```

Fig. 3, 4, 5 e 6: Gramática

Estruturas de dados

Para criar a tabela de símbolos, usamos também uma lista ligada onde cada nó (Fig. 7, 8 e 9) representam um elemento da árvore e está ligado aos seus filhos e irmãos, seguindo assim a estrutura definida no enunciado. Os elementos da tabela de símbolos podem ser de 4 tipos (variável global, variável local, classe, função). As estruturas da Fig. 8 e 9 representam os nós no caso de serem variáveis e funções, respetivamente. No caso de ser classe, é apenas identificado o seu tipo e as outras estruturas não são utilizadas.

Na criação das tabelas percorremos a árvore de sintaxe abstrata e adicionamos as funções e variáveis globais à tabela global. No caso de a função ou a variável global ser repetida, atribuímos o valor repetido a 1, de modo a não aparecer duas vezes na impressão das tabelas, aplicando-se o mesmo processo às variáveis locais das funções.

```
typedef enum {var, func, class, fielddecl} func_type;

typedef struct table_element {
    func_type decl_type;            // Diz qual o tipo do elemento em causa
    char * id;                      // Nome
    funcdecl * funcdecl;            // Pointer para funcao
    vardecl * vardecl;              // Pointer para variavel
    struct table_element * next;    // Pointer para o proximo no da tabela
    int line;                       // Nr da linha (erros - nao estao feitos)
    int column;                     // Nr da coluna (erros - nao estao feitos)
    int repetido;                   // Flag se esta repetido na tabela class (1) ou nao (0), mexida no check_funcdecl e no check_fielddecl
} table_element;
```

Fig. 7: Estrutura de dados da tabela de símbolos

```
typedef struct _t1 {
    char * type;                    // Tipo da variavel
    char * param;                   // Se e parametro de entrada ou local
} vardecl;
```

Fig. 8: Estrutura de dados da tabela de símbolos caso uma entrada seja uma variável global ou local

```
typedef struct _t2 {
    char * type_return;             // Tipo de retorno da funcao
    struct table_element * vars;    // Pointer para as variaveis que a funcao tem
    int n_params;                   // Nr total de parametros de uma funcao
    int n_params_header;            // Nr total dos parametros de entrada (MethodHeader)
    int parametros_iguais_todos;    // Flag se esta tabela dos methods (1) ou nao (0) mexida no final do check_func_decl
} funcdecl;
```

Fig. 9: Estrutura de dados da tabela de símbolos caso uma entrada seja uma função

Para criar a árvore AST, usamos uma lista ligada onde cada nó (Fig. 10) representa um elemento da árvore e está ligado aos seus filhos e irmãos, seguindo assim a estrutura fornecida no enunciado.

Para criar a árvore de sintaxe abstrata anotada, adicionámos um parâmetro em cada nó nomeado de *annotation*, evitando deste modo a construção de uma nova árvore. Para adicionar a anotação aos nós da lista ligada, percorremos a árvore de novo com o auxílio da tabela de símbolos de cada função, adicionando assim a anotação correta de cada nó. Em alguns casos, os ID's não se encontram na tabela individual das funções, pelo que temos de percorrer a tabela global de modo a adicionar a anotação correta. Na chamada de métodos,

caso a função com parâmetros inseridos não exista, substituímos por um método compatível com esses parâmetros.

```
struct node{
    char* type;                         // Tipo da variavel (boolean, int, double)
    char* value;                        // Valor da variavel
    struct node *dad;                   // Indica o pai
    struct node *bros;                  // Indica os irmaos
    struct node *childs[MAX_CHILD];     // Array com os filhos (usado para dar print)
    int index_childs;                   // Nr Total de Filhos
    char* annotation;                   // Anotacao para a arvore anotada
    int line;                           // Nr da linha (erros - nao estao feitos)
    int column;                         // Nr da coluna (erros - nao estao feitos)
};
```

Fig. 10: Estrutura de dados da AST

Geração de código

Nesta etapa, a árvore AST é percorrida mais uma vez e, com o auxílio das tabelas criadas, imprimimos o código LLVM, correspondente a cada operação. As tabelas vão servir para distinguir se as variáveis a utilizar são locais (*%%.%s*) ou globais (*@%s*).

Para evitar que as variáveis globais tenham o mesmo nome que as variáveis locais, optámos pela implementação do nome do seguinte modo *@nome_tipoDaVariável*. No caso das funções, para evitar que estas tenham o mesmo nome, seguimos uma sintaxe semelhante *@method_nomeMétodo_retornoDoMétodo_tipoDosParâmetros*.

Para identificar os casos que não possuem a função *main*, utilizámos uma *flag* que, ao encontrar uma função com o mesmo nome, altera o seu estado para 1. No caso da *flag* ser 0, é necessário criar uma função *main*, pois o programa não possuí e necessita da mesma para funcionar.

Nos casos de *assignments*, ao atribuir um *int* a um *ID* do tipo *double*, é necessário passar o *int* para *double*, de modo a não ocorrer erros. O restante código das expressões e *statements* tiveram como base a documentação do LLVM.

Comentário

O nosso compilador teve pontuação máxima no Mooshak nas duas primeiras metas. Na terceira meta, implementámos a parte essencial para prosseguir para a meta 4, no entanto, não implementámos corretamente os erros (os que fizemos, estão comentados no código, de modo a não afetarem a pontuação). Na quarta meta, implementámos partes variadas, conseguindo assim pontos um pouco por todas as categorias.