Relatório do Projeto de Compiladores 2020/21

Compilador para a linguagem UC

Gabriel Mendes Fernandes Nº 2018288117 email: gabrielf@student.dei.uc.pt
Pedro Miguel Duque Rodrigues Nº 2018283166 email: pedror@student.dei.uc.pt

1. Escrita da gramática da linguagem

Na re-escrita da gramática da linguagem fornecida em notação EBNF foram efetuadas algumas modificações com o objetivo de tornar-se possível a sua implementação na ferramenta yacc sem a existência de quaisquer ambiguidades que se manifestam sobre a forma de conflitos “shift-reduce” e “reduce-reduce”.

A primeira opção tomada com o fim de remover ambiguidades foi a introdução de regras que determinam a associatividade e precedência de certos operadores em relação a outros. A associatividade é introduzida no yacc através dos comandos “%left”, “%right” e “%nonassoc”. A precedência é especificada seguindo a regra que diz que “a declaração de associatividade se encontre abaixo de outra terá maior precedência”.

A tabela seguinte apresenta a precedência implementada e associatividade na nossa gramática e que segue as especificações do standard de C99.

Operador	Associatividade	Precedência
COMMA	%left	15
ASSIGN	%right	14
OR	%left	13
AND	%left	12
BITWISEOR	%left	11
BITWISEXOR	%left	10
BITWISEAND	%left	9
EQ, NE	%left	8
GT, GE, LT, LE	%left	7
MINUS, PLUS	%left	6
MUL, DIV, MOD	%left	5
NOT	%right	4

A segunda opção tomada e relativa ainda à precedência de operadores é no que diz respeito aos operadores unários e a construções if-then-else que podem, segundo a linguagem omitir o “else”. De acordo com as regras de precedência foram descritas anteriormente foram introduzidos operadores com as seguintes regras de associatividade como auxiliares.

A tabela seguinte apresenta a precedência e associatividade dos operadores auxiliares:

Operadores	Associatividade	Precedência
NO_ELSE	%nonassoc	3
ELSE	%nonassoc	2
UNARY_OPERATOR	%nonassoc	1

No caso deste operadores auxiliares eles são usados recorrendo à instrução “%prec” do yacc que permite atribuir a uma dada regra da gramática uma precedência definida por uma "variável". Desta forma, conseguimos atribuir às regras que tenham operadores unários a maior precedência possível na linguagem e nos fluxos condicionais dar prioridade à regra if-else em relação à mesma sem o “else”.

A terceira opção tomada na construção da nossa gramática foi a utilização das instruções %destructor que o yacc fornece para limpar a sua stack no caso da ocorrência de erros sintáticos (ou seja situações onde não existe um matching possível e caímos na classe pré-definida “error”).

As restantes modificações efetuadas na gramática apenas dizem respeito:

1. À implementação de ciclos que permitam produzir listas de expressões, declarações ou definições. Nestes casos a implementação foi realizada maioritariamente através produções recursivas à direita na nossa gramática.
   Por exemplo:

   ```
   Declaration: TypeSpec Declarator DeclaratorList SEMI
              | error SEMI
              ;

   DeclaratorList: COMMA Declarator DeclaratorList
                 |  /* epsilon */
                 ;
   ```

2. A regras que permitam lidar com situações onde uma dada construção da gramática é opcional.
   Por exemplo:

   ```
   RETURN ExprList SEMI
   RETURN SEMI
   ```

2. Algoritmos e estruturas de dados utilizados

Os algoritmos utilizados são maioritariamente algoritmos de pesquisa e travessia das estruturas de dados utilizadas para a implementação deste compilador, sendo estas listas ligadas, vetores e árvores.

De facto, para implementar a árvore de sintaxe abstrata (AST) começamos por criar um modelo/objeto que facilitasse a inserção e a manutenção de nós nesta árvore que tem de ser m-ária para poder representar corretamente a estrutura de um programa em "UC". Apresenta-se de seguida o modelo/objeto criado (recorrendo a uma "struct") para representar/guardar a informação sintática necessária e uma imagem que ilustra a forma de como essa informação está organizada na estrutura de dados.

```
typedef const char *type_t;
```

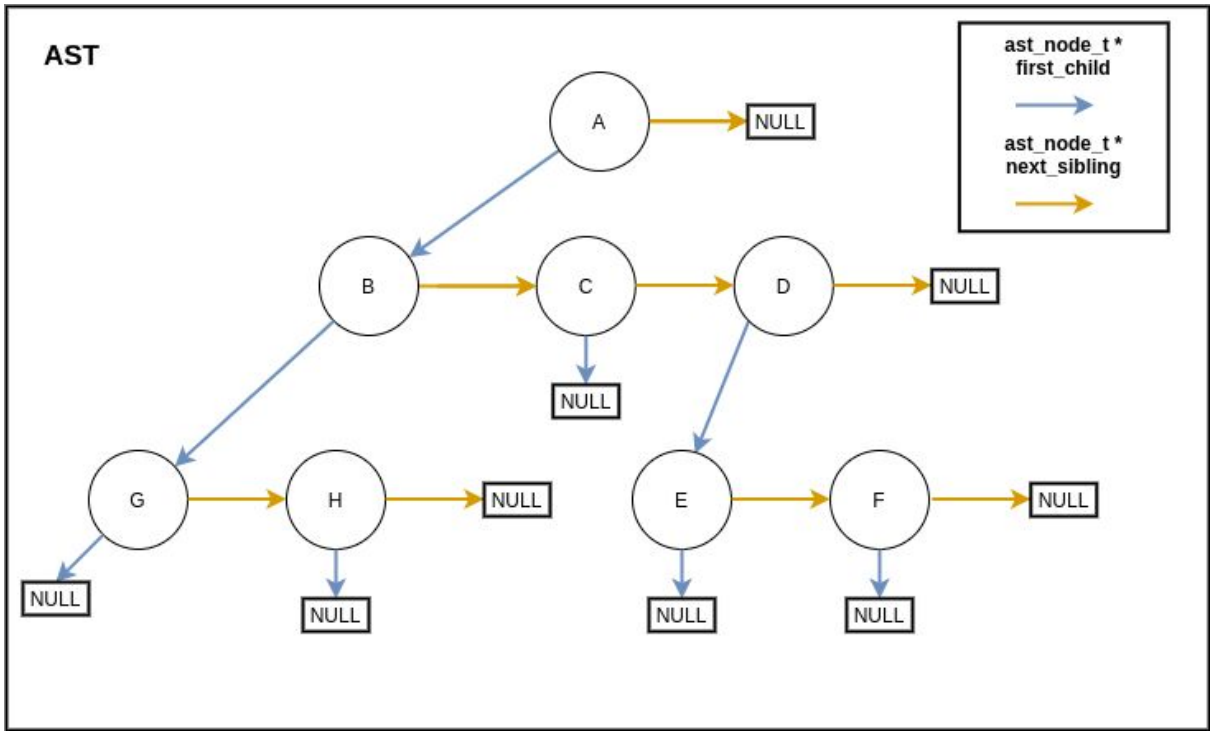
```
typedef struct Annotation annotation_t;
struct Annotation {
    type_t type;
    param_t *parameters;
};

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

typedef struct ASTNode ast_node_t;
struct ASTNode {
    const char *id;
    token_t token;
    annotation_t annotation;
    ast_node_t *first_child;
    ast_node_t *next_sibling;
};
```

- struct ASTNode (ast_node_t) : estrutura de dados de suporte da AST
  - const char *id: Nome do nó da árvore e.g "FuncDefinition", "Declaration", …
  - Token (token_t) token: Token que é emitido aquando a análise lexical
    - char *value: valor do token capturado (yytext)
    - int line: Linha onde o token foi identificado
    - int column: Coluna onde o token foi identificado

- Annotation (annotation_t) annotation: estrutura que guarda informação do nó aquando a sua anotação durante a análise semântica.



- ast_node_t * first_child: Ponteiro para o primeiro nó filho de um dado nó
- ast_node_t * next_sibling: Ponteiro para o irmão de um dado nó ou seja o seguinte filho de um nó pai.

Quanto à tabela de símbolos optamos por utilizar uma estrutura de dados linear pelo facto de termos restrições quanto à ordem que as tabelas de símbolos das várias funções tinham que ter. Assim criamos também modelos/objetos recorrendo a estruturas (“struct”), para representar uma tabela de símbolos e um símbolo que essa tabela irá conter, juntamente com toda a informação auxiliar que achamos necessária para poder completar a análise semântica de uma forma mais fácil e eficiente, tendo em mente todas as restrições impostas sobre a ordem que era necessária ter no output do nosso compilador.

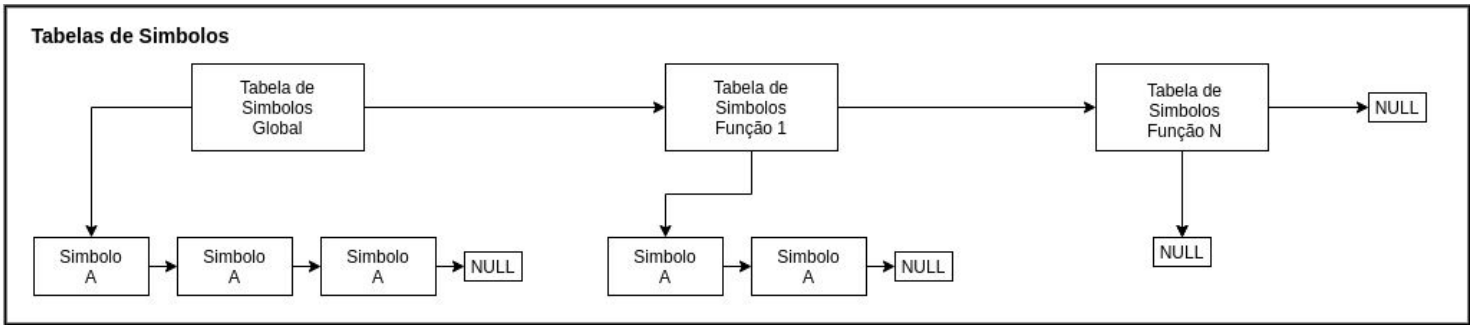
Apresenta-se de seguida o modelo/objeto criado (recorrendo a uma “struct”) para representar/guardar a informação necessária para a análise semântica e uma imagem que ilustra a forma de como essa informação está organizada na estrutura de dados.

```
typedef struct SymbolTable symtab_t;
struct SymbolTable {
    const char *id;
    bool is_defined;
    sym_t *symlist;
    symtab_t *next;
};
```

```
typedef struct Symbol sym_t;
struct Symbol {
    const char *id;
    int llvm_var;
    bool is_param;
    bool is_defined;
    type_t type;
    param_t *parameters;
    sym_t *next;
};
```

- SymbolTable (symtab_t): Estrutura de dados representativa de uma tabela de símbolos.
    - const char *id: Nome da tabela de símbolos (nome da função)
    - bool is_defined: Representa o estado da tabela. Caso uma função esteja apenas declarada terá o valor lógico false. A partir do momento em que ocorre uma definição dessa função torna-se true esse valor.
    - (struct Symbol) sym_t *symlist: Ponteiro para uma lista de símbolos que essa função tem na sua scope.
    - symtab_t *next: Ponteiro para outra tabela existente no nosso programa caso ela exista. Assumimos sempre que a tabela global existe para qualquer programa e que a cada definição as tabelas são mantidas numa estrutura ligada.

- Symbol (sym_t) : Estrutura de dados representativa de um símbolo existente numa tabela de símbolos no nosso programa.
    - const char *id: Nome do símbolo que foi declarado/definido numa dada tabela.
    - int llvm_var: Number associado a uma dada variável aquando a geração de código llvm.Todas os simbolos terão um número único dentro de uma dada função visto que o llvm funciona sobre o princípio do SSA (Static Single Assignment).
    - bool is_defined: Representa o estado do símbolo. Caso um símbolo esteja apenas declarado terá o valor lógico false. A partir do momento em que ocorre uma definição de um dado símbolo torna-se true este valor.
    - type_t type: Return type do símbolo caso este seja função ou então type do símbolo caso se trate de uma variável.
    - (struct Parameter) param_t parameters: Lista ligada de parâmetros no caso de o símbolo ser uma função. Caso contrário este valor permanece a NULL.
    - sym_t *next: Ponteiro para o próximo símbolo na lista ligada de símbolos.



Em termos de algoritmos utilizados estes são maioritariamente algoritmos de travessia de listas onde se destacam a procura e a inserção de nós. Para a inserção ser conseguida percorremos a lista até ao fim e fazemos o último nó apontar para o que está a ser inserido. No caso da pesquisa é a mesma ideia, mas paramos assim que encontramos o nó desejado, devolvendo true caso ele exista e false caso contrário. Estes algoritmos podem ser visualizados nas nossas funções que tem o prefixo “add”.

No caso da AST visto que é uma árvore construída à base de estruturas lineares a inserção de elementos nesta é a mesma que fazemos numa lista ligada. No entanto destacamos que a travessia desta que é efetuada na sua construção, na análise semântica e na geração de código é realizada “em ordem” (DFS).

3. Geração de Código LLVM

Passamos agora à descrição do processo de geração de código para cada um dos nós da nossa árvore que em conjunto com todos os outros permite-nos alcançar a representação LLVM IR do nosso programa.

Para conseguirmos referenciar todas as variáveis que são declaradas dentro de uma scope usamos uma variável global que incrementamos para as variáveis dentro das funções, este counter é colocado a zero quando entramos numa nova função.
Temos,também, uma função geral para todos os operadores que depois, consoante o operador, chama a específica para cada tipo.

Processo de geração de código:

Declaration - alocamos o espaço para essa variável e atribuímos-lhe um inteiro único que fica guardado na tabela de símbolos. Se tiver um valor associado o assign desse valor é efetuado a seguir ao alloca. Levamos em consideração se estamos dentro ou fora da scope de uma função aquando da declaração da variável, se for global é-lhe atribuído um nome na forma %<var_name>, sendo var_name o nome da variável na árvore anotada.

Store - usando o nome do nó da árvore anotada, vamos à tabela de símbolos da função atual, se não houver uma variável com esse nome, fazemos o mesmo, mas na tabela global. Obtemos o valor da expressão à direita do ‘=’ e atribuímos o seu valor a essa variável. Se a expressão tiver tipo diferente da variável (int para dar store em double), damos cast usando “sitofp” do llvm.

FuncDefinition - obtemos o tipo, o nome, o tipo dos parâmetros da árvore anotada e passamos para llvm. Aquando do cabeçalho completo, procedemos ao load dos parâmetros da função para variáveis locais à função.
Entramos no function body e traduzimos as instruções neste.
Se no function body não houver return, adicionamos um por default.

Return - Calculamos o valor da expressão do return e se for necessário damos cast deste para o tipo do return.

Arithmetic, Relational Operators - Usamos uma função para ambos, passando um parâmetro que identifica se são ou não relacionais. Nesta função fazemos load dos operandos da operação, se estes forem terminais ou usamos a variável temporária criada no llvm das operações prévias a esta. Após o load dos operandos, os operandos aritméticos vão para a sua função de print e os relacionais para a sua. Ao voltar à comum a ambos, retornamos o valor atribuído à operação executada e atualizamos numa variável global o tipo desta.

Bitwise Operators - Damos load dos operandos se forem terminais ou usamos a variável que contém as prévias operações, executamos a operação e retornamos o valor da variável onde ficou stored a operação.

Logical Operators - Damos um branch incodicional para sabermos sempre a label inicial da operação, avaliamos o operando 1, damos um branch condicional, avaliamos o operando 2. De seguida vamos para uma label final onde verificamos o valor da expressão usando o phi do llvm. Retornamos o valor da variável onde ficou stored a operação.

Unary operators - avaliamos o que está a ser afetado pelo operador unário, aplicamos-lhe o operador e retornamos o valor da variável onde ficou stored a operação.

If - avaliamos a condição do if (recorrendo a outras funções, como por exemplo a dos operadores), e consoante o resultado desta avaliação, saltamos para o ramo true ou false do if. Criamos sempre a label para o else, mesmo que este não exista. Dentro das labels do if e do else executamos as instruções contidas nestes e damos um brach incodicional para uma label final onde prosseguimos com a execução do código.

While - criamos 2 labels. Avaliamos a condição, se for true vamos para a primeira label e executamos as instruções dentro do while, no fim destas avaliamos a condição outra vez, se for true voltamos à primeira label, caso contrário segue para a segunda, saindo assim do ciclo.

Call - damos load dos parâmetros das funções para variáveis temporárias, executando quaisquer operações necessárias para obter o seu valor e fazemos o call usando estas variáveis temporárias. A função que processa o call retorna o valor da variável onde o resultado ficou stored para poder ser utilizado noutras operações ou para stores.