UNIVERSIDADE DE COIMBRA

Relatório

-

Compilador para a linguagem UC

Filipe José Good da Silva - 2015239132
Henrique Jorge Santos Branquinho - 2015239873

Junho 2018

Gramática re-escrita

Na meta anterior, fizemos a análise lexical, em que identificámos os tokens da linguagem. Contudo, é necessário garantir a ordem pela qual se organizam.

Para garantir a ordem é preciso utilizar uma gramática que indicará a ordem sintática. Para tal utilizamos a ferramenta YACC, que reconhecendo sequências de tokens que constituem o programa, verifica se obedecem à gramática.

Alterámos o analisador lexical para conseguir enviar os tokens reconhecidos para o YACC, onde efectuamos a análise sintática.

Criámos, no analisador sintático, uma union que define a estrutura de uma variável do tipo yylval.

```
%union{
    struct node* node;
    struct token * tok;
}
```

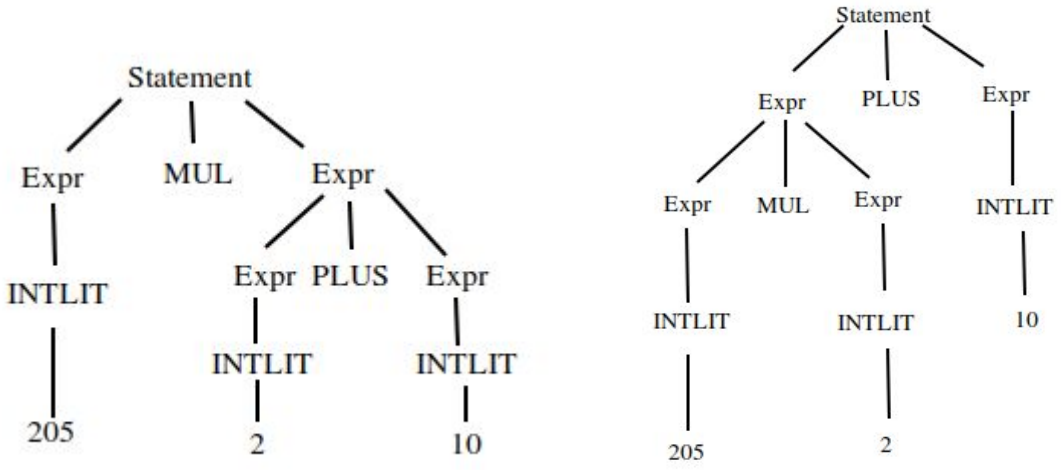
Os tokens identificados, são passados para o YACC através do campo token, que tem a seguinte estrutura:

```
typedef struct token
{
    char * value;
    int nline;
    int ncol;

}token;
```

Ambiguidade

Após analisar a gramática dada, verificámos que a mesma é ambígua. Dando um exemplo simples: dada a expressão 205*2+10, é possível reduzi-la de duas maneiras diferentes.



Como se verificou, existe ambiguidade na gramática fornecida, pois dada a mesma expressão, chegamos a dois resultados distintos. É necessário então, eliminar qualquer ambiguidade na gramática.

Alterações à gramática

A gramática fornecida, para além de conter ambiguidade, está na notação EBNF, que não é aceite pelo analisador sintático. Com isto, necessitámos de alterar a gramática.

Após analisar as regras de precedência da Linguagem C, definimos a prioridade das operações unitárias. Utilizamos %left e %right, que correspondem a associatividade à esquerda ou à direita. Utilizámos %nonassoc para o token ELSE, para indicar que este não tem qualquer associatividade.

```
%nonassoc NO_ELSE
%nonassoc ELSE

%left COMMA
%right ASSIGN
%left OR
%left AND
%left BITWISEOR
%left BITWISEXOR
%left BITWISEAND
%left EQ NE
%left LE LT GT GE
%left PLUS MINUS
%left MUL DIV MOD
%right UNARY NOT
%left RPAR LPAR
```

Foi necessário tratar dos casos de zero ou mais repetições e tokens opcionais. Segue-se um exemplo abaixo:

- Declaration→ TypeSpec Declarator {COMMA Declarator} SEMI
    - Para garantir que “COMMA Declarator” são repetidos zero ou mais vezes, criámos um novo estado responsável de repetir-se ou terminar.

```
declaration:
     type_spec aux_declaration SEMI
    | error SEMI

    ;

aux_declaration:
    declarator              {$$ = $1; }

    | aux_declaration COMMA declarator  {$$ = $1; addBro($1,$3);}
    ;
```

Mais exemplos?? Estamos a ficar sem palavras xD

Para resolver o problema de ambiguidade acima descrito, resolvemos recorrer à documentação do C99. Dividimos as diferentes operações em diferentes regras, e vamos da regra mais geral à mais específica, começando em *expr* e acabando em *primary_expression.*

Deparámos-nos com conflitos shift/reduce nos statement IF-ELSE.

Ao ler os tokens, poderá ser feito um reduce pela primeira regra, ou poderá também ser feito um shift pela segunda regra e ficar à espera de um ELSE.

Decidimos criar um novo token NO_ELSE, que seria usado nos casos em que não há else. Utilizámos a instrução %nonassoc do yacc nos tokens ELSE e NO_ELSE, e com isto, ambos ficam sem associatividade.

AST e tabela de símbolos

Para a criação da AST, foi criada uma estrutura *node*, que contém um tipo de nó (se é Id, Return, If…), um campo para o seu valor (caso seja RealLit, IntLit, ChrLit ou Id), um campo para um nó irmão , um campo para um nó filho, um campo para a anotação, e o número da linha e coluna a que diz respeito no código fonte.

```
typedef struct node
{
    struct node* children;
    struct node* brother;
    char * nodeType;
    char * value;
    char * annotate;
    int nline;
    int ncol;
}node;
```

Durante a análise sintática, são criadas estruturas deste género e passadas através da estrutura union do yylvalue, que contém um campo para um nó deste tipo. A AST é construída de "baixo para cima", sendo que o último nó a ser criado é a raiz da árvore, o nó "Program". A construção da árvore recorre a duas funções principais, **addBro** e **addChild** , que adicionam um nó como irmão ou como filho de outro nó. Caso o nó tenha mais irmãos, esta estrutura acaba por funcionar como uma lista ligada, pois são percorridos todos os irmãos até ao fim da lista, onde é adicionado o novo nó.

Para a criação da tabela de símbolos, criámos 3 estruturas:

- **sym_table** – A estrutura sym_table (tabela de símbolos) , tem um campo para uma estrutura param_list e para uma estrutura var_list, que dizem respeito, respetivamente, aos parâmetros e às variáveis declaradas na função, e um campo com um ponteiro para a próxima tabela de símbolos.. As variáveis globais e as declarações de funções são guardadas como variáveis numa tabela do tipo sym_table chamada
- **global_table**- A tabela correspondente a cada função é inserida numa lista de tabelas chamada **functions_table**. Tem também uma variável que guarda o número de parâmetros da função, e uma flag que diz se a função já foi definida ou não.
- **var_list** – A estrutura var_list (lista de variáveis) tem um campo com o tipo da variável e um campo para o seu id. Tem também uma flag para saber se diz respeito a uma função ou a uma variável, e o número de parâmetros, caso diga respeito a uma função.
- **param_list** – A estrutura param_list tem um campo com o id e o tipo do parâmetro, e também um campo para guardar o registo que lhe foi associado na parte da geração do código.

```c
typedef struct _param_list{
    char * type;
    char * id;
    int var;
    struct _param_list * next;
}param_list;

typedef struct _var_list{
    int function; //Indica se e funcao

    char * id;
    char * type;
    int nparameters;
    param_list * parameters;
    struct _var_list * next;
}var_list;

typedef struct _sym_table{
    int function; //TRUE -> Function table FALSE -> GLOBAL
    int defined; //Se ja foi definida ou nao
    char * type;
    char * id;
    int nparameters;
    var_list * variables;
    param_list * parameters;
    struct _sym_table * next;
}sym_table;
```

Declarações de funções são guardadas na **global_table**, e é criada e inserida uma nova **sym_table** que diz respeito à função declarada, na **functions_table**. Quando a função é definida, esta tabela é atualizada. Desta forma, as tabelas das funções são impressas pela ordem em que foram declaradas.

Geração de código

Na geração de código, inicialmente são declaradas as funções putchar e getchar, seguidas de todas as variáveis globais conhecidas. Após isto, é definida a função main no llvm. Esta função não corresponde à main definida no código que está a ser compilado. Nesta main, são calculados os valores das atribuições nas declarações globais (caso haja), e é feito o store deste valor na respetiva variável global.

As variáveis globais são representadas com @%s, em que %s é o nome original da variável. As variáveis locais são representadas com %%s, em que %s é o nome original da variável. As funções são representadas com @f.%s, em que %s é o nome original da função.

A main do LLVM invoca a função main (call @f.main) definida no código que está a ser compilado. Em cada função, existem três contadores: um contador de registos utilizados **(temp_vars**), e dois contadores de labels (para os if's e while's, e para os and'e or's).

Entre cada operação é feita a conversão dos operandos para o tipo da operação, através das funções **sitofp, trun, sext** e **zext**. Os resultados das operações de comparação são sempre convertidos de i1 para i32, para manter a coerência com as anotações das operaçõe (como são anotadas como int, o nó pai assume que vai receber um i32).

Para as statements **if**, é processada primeiro a condição, e o resultado é comparado com 0. São criadas duas labels (três no caso em que existe um **else**), uma com o nome **if.%d** em que %d é o contador de labels, e outra com **if.finish (**no caso do else , há também uma label if.else).

Nas statements **while**, é criada uma label inicial onde é testada a condição. De seguida, há um salto para a label do corpo do while, caso seja true, e para a label de fim do while, caso seja false. Na label do corpo do while, há um salto para a label inicial para voltar a testar a condição.

Com os **and** e **or**, foi preciso definir também três labels. No entanto, como o segundo filho de cada AND ou OR pode criar novas labels (se for and ou or), é necessário uma operação **phi** para guardar o resultado da operação dependendo da label de onde se veio.

As restantes operações são bastante simples, bastou converter código C para LLVM com clang para perceber quais as operações a usar.

Os parâmetros de cada função são associados a um registo no início da definição da função.