FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE D COIMBRA

Compilador para a Linguagem UC

Projeto de Compiladores 2020/2021

Eduardo Cruz 2018285164
Rodrigo Sobral 2018298209

Introdução

No âmbito da unidade curricular de *Compiladores* presente na *Licenciatura em Engenharia Informática da Universidade de Coimbra*, iremos implementar um compilador para uma linguagem baseada em *C*, nomeadamente *UC*.

Neste relatório iremos apresentar uma breve explicação das nossas decisões de implementação relativamente à gramática descrita no enunciado do problema fornecido pelo professor *Raul Barbosa*, além das estruturas de dados e algoritmos usados para a construção da AST (*Abstract* Syntax *Tree*).

Assim de forma a implementar este programa recorremos ao *Lex*, um analisador léxico, e ao *Yacc*, um analisador sintático, que permite descrever a gramática e verificar a sua correta derivação. Já para programar todas as funções utilizadas na construção da árvore de sintaxe abstrata (AST), da tabela de símbolos e para a geração de código utilizámos a linguagem *C*.

Analisador Sintático - Gramática (Meta 2)

Uma vez que a gramática dada, em notação *EBNF*, é ambígua e apresenta produções opcionais que podem conter zero ou mais repetições, tivemos de recorrer a novas produções e, no caso das produções que se repetem 0 ou mais vezes, procedemos a uma recursão à esquerda, isto porque em casos de repetições mais extensas resulta numa otimização evidente.

Apresentamos agora as nossas produções opcionais em casos ambíguos, o nosso tratamento face aos vários erros e as respetivas explicações:

```
program: functionsAndDeclarations
functionsAndDeclarations: functionsAndDeclarations functionDefinition
      | functionsAndDeclarations functionDeclaration
      | functionsAndDeclarations declaration
```

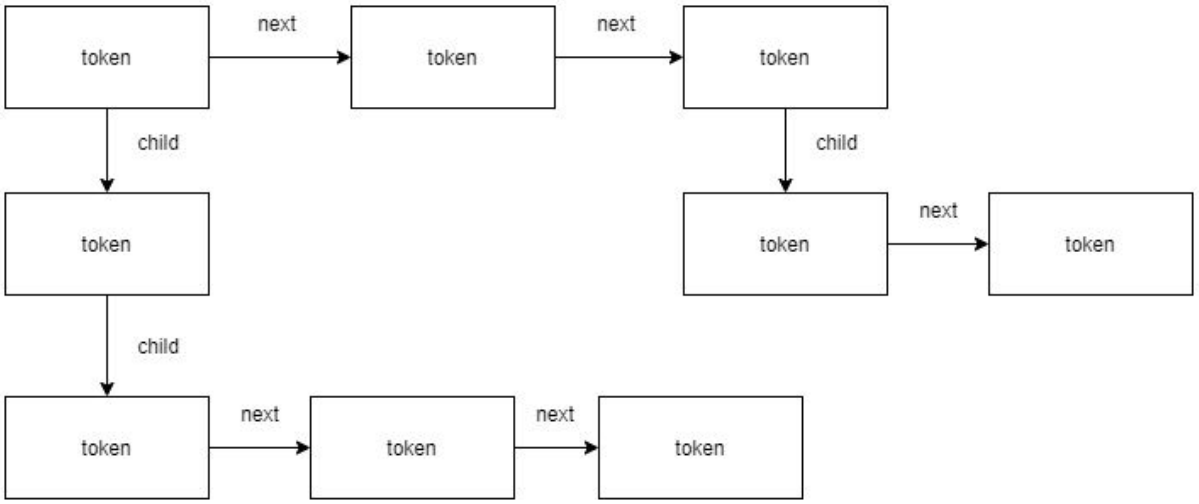
```
      | functionDefinition
      | functionDeclaration
      | declaration

functionDefinition: typeSpec functionDeclarator functionBody
functionBody: LBRACE RBRACE | LBRACE declarationsAndStatements RBRACE

declarationsAndStatements: statement declarationsAndStatements
      | declaration declarationsAndStatements
      | statement
      | declaration

declaration:  typeSpec declarator declaratorsAux SEMI | error SEMI
declaratorsAux: /*epsilon*/ | declaratorsAux COMMA declarator
declarator: ID | ID ASSIGN exprComplete

/*************************************************************************
     Desta forma, conseguimos fazer a análise indeterminada de
declarações de variáveis, tenham elas sido definidas ou não, além da
análise, também indeterminada, de declarações e definições de funções.
Temos em consideração possíveis erros sintáticos durante a análise das
expressões associadas à declaração de variáveis.
     Nestes, e em todos os casos que nos deparamos com um erro,
colocamos o topo da pilha a NULL. Isto é útil em situações onde
precisamos de saber se o resultado duma produção não é NULL para
podermos adicionar um nó child adequadamente (como é o caso das
produções declarationsAndStatements -> statement e
declarationsAndStatements -> declaration). Se tal não se verificar
então criamos o nó da leitura atual e colocamo-lo no topo da pilha
($$), depois adicionamos, se houverem, os seus childs, que serão os
argumentos adequados consoante a leitura ($1, $2, etc).
*************************************************************************/

statement:  SEMI
    |   exprComplete SEMI
    |   LBRACE statementsAux RBRACE
    |   LBRACE error RBRACE
    |   LBRACE RBRACE
    |   IF LPAR exprComplete RPAR statError %prec THEN
    |   IF LPAR exprComplete RPAR statError ELSE statError
    |   WHILE LPAR exprComplete RPAR statError
    |   RETURN SEMI
    |   RETURN exprComplete SEMI
```

```
statementsAux:  statError | statementsAux statError
statError: statement | error SEMI


/*********************************************************************
     Aqui estão presentes duas ambiguidades, uma clássica, conhecida
como Dangling else, e a error SEMI. A primeira baseia-se no facto de
uma instrução poder acabar com ; ou com {}, no entanto é facilmente
resolvida com o comando %prec THEN. Já a segunda foi para nós a
situação mais difícil de resolver e foi nesta etapa que residiu, a
nosso ver, a maior dificuldade desta meta. No entanto, a sua resolução
parece mais simples do que de facto é. Neste caso tivemos de criar uma
nova produção auxiliar statementsAux que resulta na produção
indeterminada statError. Esta, por sua vez, dá origem ao erro error
SEMI ou, em caso de não deteção de erros, um novo statement. Assim não
há qualquer tipo de ambiguidade relativamente ao error SEMI presente na
produção de declaration e todos os casos de erro serão detetados.
*********************************************************************/

expr:   expr ASSIGN expr
    |   expr PLUS expr
    |   expr MINUS expr
    |   expr MUL expr
    |   expr DIV expr
    |   expr MOD expr
    |   expr OR expr
    |   expr AND expr
    |   expr BITWISEAND expr
    |   expr BITWISEOR expr
    |   expr BITWISEXOR expr
    |   expr EQ expr
    |   expr NE expr
    |   expr LE expr
    |   expr GE expr
    |   expr LT expr
    |   expr GT expr
    |   PLUS expr %prec NOT
    |   MINUS expr %prec NOT
    |   NOT expr
    |   functionCall
    |   ID
    |   INTLIT
    |   CHRLIT
```

```
    |   REALLIT
    |   LPAR exprComplete RPAR
    |   LPAR error RPAR

exprComplete: expr | exprComplete COMMA expr

functionCall: ID LPAR RPAR
    |   ID LPAR exprList RPAR
    |   ID LPAR error RPAR

/*************************************************************************
     A leitura de expressões é feita quase que recursivamente,
prevendo mais uma vez possíveis erros dentro de parêntesis, quer na
chamada de funções, quer numa expressão. Um caso particularmente
delicado foi a associatividade à direita do NOT nos casos PLUS expr e
MINUS expr mas foi resolvido com o comando %prec NOT em ambos os casos.
*************************************************************************/
```

Árvore de Sintaxe Abstrata - Estrutura de Dados

A árvore de sintaxe abstrata implementada é uma árvore de listas ligadas, preenchida durante a análise sintática, onde cada nó tem o seu nó *filho* e o seu nó *next*. Ou seja, há uma hierarquia de alturas/profundidades na árvore, onde o nó com a profundidade 0 seria a raíz, neste caso, *"Program"*, consequentemente o seu nó *"child"* seria o nó *header* duma lista ligada onde teríamos todos os nós referentes a todas as declarações de funções (*FuncDeclaration*), variáveis (globais, *Declaration*) e definições das funções (*FuncDefinition*) e assim sucessivamente para o nós *child* de *FuncDeclaration, FuncDefinition e Declaration*. Ou seja, quanto maior a profundidade de cada nó na árvore, maior a indentação presente na sua impressão.



Da análise sintática para a semântica, tivemos de fazer um pequeno ajuste na árvore e adicionámos os dados *struct param *param_list* e *struct token *tk,* para que possamos fazer as anotações da *AST* e a indicação da linha e coluna dos erros, na meta 3. Assim, cada *token* possui, incluindo os nós *next* e *child* a seguinte informação:

```
      typedef struct node{
          char *str; /*Nome do nó, ex: FuncDefinition, Declaration*/
          struct token *tk; /* Contém a linha e coluna do nó no código
e o respetivo valor associado, ex: main, i, f1*/
          _type type; /*Tipo do nó, ex: voidlit, intlit, etc*/
          struct param* param_list; /*Lista ligada com todos os tipos
dos seus parâmetros, no caso de se tratar duma função*/

          struct node *next;
          struct node *child;
      } node;
```

Analisador Semântico - Algoritmo (Meta 3)

A construção da Tabela de Símbolos utilizada para realizar a Análise Semântica nesta fase é conseguida através da ramificação e interpretação dos nós da AST. De forma geral, começamos por percorrer iterativamente os nós de profundidade 1 na árvore

(*FuncDefinition, Declaration* e *FuncDeclaration*), de seguida, “perdemos o controlo” da leitura, ou seja, em cada um deles, trabalhamos consoante o código que encontrarmos, o que faz com que lidemos com diversos tipos de *inputs* tais como *function calls, expressions* ou *statements*.

Lidar com toda esta variedade, implica uma constante verificação dos seus tipos de dados, por isso, com o auxílio das funções *get_statement_type(node, sym_table)* e *checkConflitingTypes(_type, _type)* ou *check_params_list_types(sym*, sym*)*, conseguimos prever qualquer situação de conflito de tipagens, quer nas variáveis, quer nas funções, ou até mesmo em expressões onde um operador não pode ser aplicado a um certo par de tipos.

Outro tipo de situações exigem que nós consultamos as tabelas de símbolos, até ao momento armazenadas, para que saibamos se a leitura que estamos a fazer é, por exemplo, duma função já definida (*isVarNameInSymList(char*, sym_table*)*), ou, por outro lado, dum símbolo não declarado, e por isso, desconhecido, nestes casos disparamos os erros indicados para cada situação, descritos no enunciado fornecido, com a informação correspondente presente na árvore *AST* (valor(es), linha e coluna).

Em casos livres de erros e prontos para serem adicionados às tabelas de símbolos, trabalhamos com as funções presentes no ficheiro *symbol_table.c/h*. Estas funções são maioritariamente recursivas e baseiam-se em criadores de símbolos e parâmetros, *adders/getters* de símbolos e tabelas, conversores *_type->string*, *string->_type, printers*, pequenas verificações relacionadas às tabelas e libertações de memória. Desta forma o código fica mais organizado pela área em que ele atua.

Tabela de Símbolos

Relativamente às tabelas de símbolos, há dois tipos de tabelas: a *Global Symbol Table*, onde são impressas todas as declarações (nós *Declaration* e *FunctionDeclaration*) corretamente feitas, e as tabelas de símbolos relativas a todas as funções corretamente definidas. Estas por

sua vez possuem: o tipo de dado que a respetiva função retorna, os tipos e identificadores de todos os parâmetros, e todas as declarações de variáveis de escopo, respetivamente.

Para armazenar toda a informação necessária nas tabelas de símbolos recorremos a uma lista ligada na qual, cada nó tem a seguinte informação:

```
typedef struct sym_table{
    char* name; /*Nome da Tabela, ex: Global, main, etc*/
    int isDef; /*Flag: 0 se a função foi declarada, 1 se foi definida*/
    struct sym *sym_list; /*Lista ligada de todos os símbolos contidos
na tabela*/
    struct sym_table *next; /*Próxima tabela de símbolos*/
} sym_table;
```

onde cada símbolo é novamente uma lista ligada com:

```
typedef struct sym {
    char *name; /*Nome do símbolo, ex: putchar, return, etc*/
    _type type; /*Tipo do símbolo, ex: voidlit, intlit, etc */
    struct param* param_list; /*Lista ligada com todos os tipos dos
seus parâmetros, no caso de se tratar duma função*/
    int isFunc; /*Flag: 0 se for variável, 1 se for função*/
    int isParam; /*Flag: 1 se for um parâmetro, 0 não se for*/
    struct sym *next; /*Próximo símbolo*/
} sym;
```

Fomos adicionando campos à medida que íamos precisando deles, contudo, tentámos reduzir ao máximo a sua quantidade de forma a que pudéssemos distinguir o máximo de variantes com o mínimo de informações possíveis.

Abaixo apresentamos um exemplo duma *Global Symbol Table* e duma *Function main Symbol Table* correspondentes ao seguinte trecho de código:

Input:

```
int i;
char c;
short s;
```

```
int main(int i1, char c1, short s1) {
    int i2; char c2; short s2;
    return 5;
}
```

Output:

```
===== Global Symbol Table =====
putchar     int(int)
getchar     int(void)
main        int(int,char,short)
c     char
s     short
i     int

===== Function main Symbol Table =====
return      int
i1    int   param
c1    char  param
s1    short param
i2    int
c2    char
s2    short
```

Gerador de Código (Meta 4)

A geração de código é feita com base num código, livre de erros, o que significa que, nesta meta, o desafio não está em prever erros mas sim em gerar código intermédio de máquina, executável.

Por definição, tivemos de tomar uma série de decisões que foram, a nosso ver, as mais indicadas dadas as diversas situações, por isso iremos listar aqui algumas delas:

- É automaticamente feita uma declaração das duas funções predefinidas: getchar() e *putchar(i32)*.

- No escopo global não é possível fazer operações de *alloca, store, add,mul,* etc, pelo que todas as variáveis globais inicializadas em *UC*, são na verdade inicializadas dentro duma função, *@_INIT_GLOBAL_VARS()*, que é sempre chamada no início da função main().

- Em *LLVM* todas as funções definidas têm obrigatoriamente um ***return***, o que significa que caso tenhamos uma função do tipo ***void***, ***int*** ou ***double***, por exemplo, e não seja escrito um ***return***, então a função retorna ***void***, **0** e **0.0**, respetivamente.

- Se uma variável não for inicializada então, por omissão, inicializa-mo-la a **0** (no caso de ser ***int***).

- O tipo de dado ***double***, em *LLVM*, é obrigado a ter uma parte decimal, o que significa que é possível escrever **.23** na linguagem *UC* mas não em *LLVM*, portanto, nestes casos, adicionamos sempre um ***0*** antes do ponto.

- Em *LLVM* uma função ou é definida ou é declarada, portanto, a menos que tenhamos apenas uma função declarada, e, nesse caso, implementamos a declaração, o que fazemos em *LLVM* é sempre a definição da função.

Outros casos muito particulares e importantes que tivemos de prever foram os seguintes:

- A propriedade absorvente das operações lógicas. Isto é, ao ter o operador lógico *and* (***&&***), se o resultado da expressão do lado esquerdo for 1, não podemos ignorar o resultado da expressão do lado direito, no entanto, se o contrário ocorrer (*0**&&**1*), não só podemos como devemos ignorar o que está do lado direito, uma vez que independentemente do que seja, o resultado final já será **0**. O oposto de todo este processo se aplica ao operador lógico *or* (**||**).

- Fazer a alocação de memória dentro da condição de um *while* faz com que o resultado fosse um *Segmentation Fault*, portanto nós tivemos de alocar previamente a memória antes de entrar no ciclo. No caso dos *ifs* não houve este tipo de problema porque trata-se apenas de uma execução de código.

- Não se fazem operações com octais em *LLVM* o que faz com que tenhamos de converter todos os números octais para decimais.

- E por fim, mas não menos importante, tivemos de ter em consideração a flexibilidade de *casting* dos tipos de dados. Para converter um ***double*** para ***int***, por exemplo, retiramos apenas a parte decimal do ***double*** em questão. O oposto seria acrescentar-lhe a parte fracionária (*(int)2.6 = 2; (double)2 = 2.0*).