Universidade de Coimbra

Faculdade de Ciências e Tecnologia

Licenciatura em Engenharia Informática

Compiladores – 2020/2021

3º Ano – 1º Semestre

Relatório do Projeto

Coimbra, 21 de dezembro de 2020

2018298731	João Pedro Pacheco Silva	joaopedro@student.dei.uc.pt
2018297934	Luís Carlos Lopes Loureiro	uc2018297934@student.uc.pt

Gramática reescrita

1. **Recursividade à esquerda:**

Numa primeira faze começamos por pegar nas produções com recursividade à direita e reescrevemo-las para que passassem a ter recursividade à esquerda.

Antes:

FunctionsAndDeclarations → (FunctionDefinition | FunctionDeclaration | Declaration) [FunctionsAndDeclarations]

DeclarationsAndStatements → (Statement | Declaration) [DeclarationsAndStatements]

Declaration → TypeSpec Declarator {COMMA Declarator} SEMI

Depois:

FunctionsAndDeclarations → [FunctionsAndDeclarations] (FunctionDefinition | FunctionDeclaration | Declaration)

DeclarationsAndStatements → [DeclarationsAndStatements] (Statement | Declaration)

Declaration → Dec SEMI

Dec → TypeSpec DeclaratorEnum

DeclaratotEnum → DeclaratorEnum COMMA Declarator

DeclaratorEnum → Declarator

Nota:

Por engano nosso acabamos por escrever Declarator COMMA DeclaratorEnum, mas o nosso intuito era fazer como está em cima.

Uma vez que o *yacc* faz analise sintática ascendente, gramáticas com recursividade à direita aumentam o risco de esgotamento da memória, dado que é necessário aplicar *shifts* repetidamente até ao fim da sequência, deixando que todo o input vá parar à pilha, para depois se consegui fazer os *reduces*. Gramáticas com recursividade à esquerda, não apresentam esse problema, pois vão fazendo *reduces* á medida que avançam na sequência.

2. **Conflitos *Shift-reduce*:**

Para resolver os conflitos Shift reduce causados pelos operadores das equações simplesmente definimos as precedências e a associatividade usando os seguintes comandos:

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

No conflito:

Expr → Expr (ASSIGN | COMMA) Expr
Expr → ID LPAR [Expr {COMMA Expr}] RPAR

Reescrevemos para:

Expr → Expr (ASSIGN | COMMA) Expr
Expr → ID LPAR [Expr] RPAR

No conflito:

Statement → IF LPAR Expr RPAR Statement [ELSE Statement]

```
IfElse: ELSE Statement          %nonassoc Y
        | %prec Y/*empty*/      %nonassoc ELSE
        ;
```

3. **Outros**:

Por fim tivemos que resolver o problema causado pelo facto dos *tokens* PLUS e MINUS terem associatividade e prioridades diferentes consoante o contexto e para isso usamos o seguinte comando:

```
Minus: MINUS Expr %prec NOT
Plus: PLUS Expr %prec NOT
```

O comando %prec faz com que a sequencia à esquerda tenha a mesma prioridade e precedência que o *token* à direita.

Algoritmos e estruturas de dados da AST

1. Construção da AST:

Para construirmos a nossa AST, criámos uma *struct* diferente para cada tipo de nó, cada uma guardando os dados necessarios. Para os nós em que se podia seguir por caminhos diferentes, usamos o atributo "type" para saber qual o próximo nó que teríamos de aceder:

```
typedef struct _26 {
        char* type;
        Function_Definition* fdef;
        Function_Declaration* fdec;
        Variable_Declaration* vdec;
} Fad;
```

```
Fad: FunctionDefinition         {$$ =  insert_Fad("fdef", $1, NULL, NULL); count++;}
        | FunctionDeclaration   {$$ =  insert_Fad("fdec", NULL, $1, NULL); count++;}
        | Declaration           {$$ =  insert_Fad("vdec", NULL, NULL, $1); count++;}
        ;
```

Para os nós em que era necessário guardar a linha e coluna para as mensagens de erro da análise semântica usamos os atributos l e c:

```
typedef struct _s19 {
        int l;
        int c;
        char* tipo;
        char* type;
        Expression* ex1;
        Expression* ex2;
}  Operator;
```

Dados como os nomes e tipos das variáveis, funções e parâmetros eram guardados usando os atributos "id" e "TypeSpec" respetivamente.

```
typedef struct _s7 {
        int l;
        int c;
        char* TypeSpec;
        char* id;
        struct _s7* next;
} ParamDeclaration;
```

A inserção dos nós segue sempre o mesmo padrão. Para cada um existe uma função que cria o nó, preenche os seus atributos com os dados recolhido e retorna o nó. Esse mesmo nó é passado como parâmetro da função *insert* do nó pai para depois ligar os dois.

```
Fad* insert_Fad(char* type, Function_Definition* fdef, Function_Declaration* fdec, Variable_Declaration* vdec)
{
        Fad* node = (Fad*)malloc(sizeof(Fad));

        node->type = type;
        node->fdef = fdef;
        node->fdec = fdec;
        node->vdec = vdec;

        return node;
}
```

2. **Percorrer a arvore:**

Para percorrer e imprimir a arvore criamos também uma função para cada nó. Cada função imprime os dados do respetivo nó e chama a função print do nó filho.

```
void print_FuncBody(int i, FuncBody* node, int option)
{
        if(node==NULL) return;
        if(option){print_ponto(i); printf("FuncBody\n");}

        i+=2;
        print_Declarations_And_Statements(i, node->list, option);
        //free(str2);
        free(node);
}
```

Algoritmos e estruturas de dados da tabela de símbolos

1. **Estruturas Usadas:**

Para a construção da tabela de símbolos, nos criamos uma estrutura que contem 3 ponteiros. Um chamado “list”, que aponta para o inicio da lista de ambientes, outro chamado “current” que aponta para o ambiente atual e outro chamado “last” que aponta para o ultimo nó da lista de ambientes e serve exclusivamente para tornar a inserção mais rápida.

```
typedef struct list4 {
        environment_node* list;
        environment_node* current;
        environment_node* last;
        int erro;
} environment_list;
```

Cada nó da lista de ambientes é representado com uma estrutura que contem um atributo chamado “defined”, que diz se a função correspondente já foi definida ou não, um atributo “env”, que contem o id da função e o tipo que ela retorna; um atributo “list”, que contem as suas variáveis e parâmetros, um atributo “next” que aponta para o próximo nó da lista e um atributo “previous” que aponta para o ambiente anterior (serve para restaurar o ambiente anterior e não para retroceder na lista)

```
typedef struct list3 {
        int defined;
        environment env;
        declaration_list* list;
        declaration_list* last;
        struct list3* next;
        struct list3* previous;
} environment_node;
```

```
typedef struct list_item2 {
        char* nome;
        char* ret;
} environment;
```

2. Gestão dos Ambientes

Para construirmos os ambientes, nós optamos por usar tabelas de dispersão, onde basicamente temos um array de listas ligadas.

Cada nó é representado por uma estrutura com um atributo “dec”, que contem o id e o tipo do símbolo e diz se é um parâmetro ou não, um atributo “env” que aponta para o ambiente onde o símbolo foi declarado, um atributo “function”, que caso o símbolo seja uma função, aponta para o ambiente que a define e um atributo “next” que aponta para o próximo nó da lista.

```
tabela= (declaration_node**)malloc(len*sizeof(declaration_node*));
```

```
typedef struct list1 {
        declaration dec;
        environment_node* env;
        environment_node* function;
        struct list1* next;
} declaration_node;
```

```
typedef struct list_item {
        int notified;
        char* nome;
        char* tipo;
        int param;
} declaration;
```

Sempre que há uma declaração é usada uma função de *hash* sobre o id e esta por sua vez calcula o índice onde se deve inserir o símbolo. Se o índice estiver vazio cria-se um novo nó, caso não esteja, é visto o ambiente a que pertence. Se não pertencer ao ambiente atual cria-se um novo nó que passará a ocupar a primeira posição da lista, caso pertença, a tabela mantém-se igual.

Para restaurar o ambiente anterior percorremos a lista de variáveis do ambiente atual. Cada nó dessa lista tem um ponteiro para o próximo nó e um ponteiro para o nó na tabela de dispersão.

```
typedef struct list2 {
        declaration_node* dec;
        struct list2* next;
        struct list2* previous;
} declaration_list;
```

Em cada nó acede-se à posição correspondente da tabela de dispersão e remove-se o símbolo no topo da lista.

Obtenção dos tipos:

Os tipos guardados no atributo “tipo” dos nós da AST são obtidos consultado a tabela de dispersão caso sejam nós que representem ids, são obtidos usando o atributo type” caso sejam nós que representem terminais, ou obtidos com base nos nós filhos caso sejam expressões.

geração de código

O compilador gera o código percorrendo a àrvore de sintaxe anotada. Este vai imprimindo conteúdo diferente consuante o nó em que se encontra.

Para traduzir o tipo simplesmente consultamos o atributo “tipo” do nó e escrevemos double para caso seja do tipo double e i32 para os restantes.

```
if(node->tipo[0]=='i' || node->tipo[0]=='c' || node->tipo[0]=='s'){
if(!strcmp(node->type, "Store")){ printf("%sstore i32 %s%d, i32* ", str, "%", *OpNumber); ConvertId(node->ex1->id); printf("\n");}
```

Para declarações de funções simplesmente escrevemos o id precedido pelo caracter @ e pelo tipo e em seguida escrevemos os tipos e ids dos seus parâmetros entre parenteses. (Para definições o processo é análogo)

EX:

int soma(int a, int b) → i32 @soma(i32 a, i32 b)

Para cada variável declarada, criamos um ponteiro do tipo correspondente e damos-lhe o mesmo id precedido pelo caracter %.

Ex:

int a → %a = alloca i32, i32 1

Procedendo deste modo conseguimos contornar o problema do LLVM ser SSA *(Static Single Assignment).* Assim sempre que seja necessário alterar o valor de a basta escrever o comando *store*.

Ex:

a = 3 → store i32 3, i32* %a

Caso fosse necessário obter o valor da variável escrevíamos o comando *load*.

Para podermos representar as expressões tivemos que manter um contador que nos indicasse o número da operação atual. Como só podíamos realizar uma operação de cada vez teríamos primeiro que realizar as operações dos nós filhos e guardar os seus valores intermédios para depois realizar as operações dos nós pais.

```
Convert_Expression(str, node->ex1, 1, OpNumber);
exp1 = *OpNumber-1;
Convert_Expression(str, node->ex2, 1, OpNumber);
exp2 = *OpNumber-1;
```

Para isso usamos um ponteiro “OpNumber” como parâmetro da função que percorre o nó filho. Este será incrementado pelas operações dos nós filhos e o seu valor é guardado após a chamada da cada função.

Esses valores são usados para saber onde estão guardados os resultados das operações anteriores. É usado o mesmo processo para saber onde estão os valores dos parâmetros quando se chama uma função.

Para traduzir os *ifs* e *elses* usamos *labels* e saltos condicionais. O nome dos *labels* têm sempre o mesmo prefixo e é usado um segundo contador para diferenciá-los.

```
printf("%s%s%d = icmp ne i32 0, %s%d\n", str, "%", *OpNumber, "%", *OpNumber-1);
printf("%sbr i1 %s%d, label %sthen%d, label %selse%d\n\n", str, "%", *OpNumber, "%", i ,"%", i);
```

Por fim os *whiles* são representados com um *label* com o prefixo “while”. No final desse mesmo *label* existe um salto incondicional que nos leva de volta ao início e a única saída é pelo salto condicional escrito no inico do *label*.