COMPILADORES

Compilador da Linguagem Juc

2019/2020

Departamento de Engenharia Informática
FCTUC

João Monte Monteiro – uc2016244006
Luis Freitas – uc2017264549

1) Gramática re-escrita

Estruturas no YACC

A única alteração feita no LEX foi a implementação das flags necessárias para envolver os erros semânticos, a AST e a tabela de símbolos.
Já no YACC, houve necessidade de reorganizar e criar estruturas auxiliares para guardar as informações sobre os tokens. A estrutura “union” tem um ponteiro para o nó da AST, para a estrutura auxiliar “type_struct” ,que contém a linha, coluna e tipo do token, e por fim, um ponteiro para a estrutura “**args**” (onde o valor do token é guardado caso seja literal).

```
struct type_struct{
        int line,col;
        const char*type;
}
;
```

Fig.1

```
%union {
        struct n* no;
        struct type_struct* t;
        struct _args* args;
};
```

Fig.2

Alguns operadores causam ambiguidade nas operações e foi necessário atribuir precedência para que cada regra seja interpretada da maneira correta. Esta precedência foi implementada na ordem que se segue e com a respetiva associatividade:

1. %left COMMA
2. %right ASSIGN
3. %left OR
4. %left AND
5. %left LT GT GE LE EQ NE
6. %left XOR LSHIFT RSHIFT
7. %left PLUS MINUS
8. %left STAR DIV MOD
9. %right NOT

Repetição / Opção:

A gramática no yacc foi alterada no que toca a regras que contêm tokens repetidos ou opcionais. Sempre que acontece um caso deste género, divide-se a regra em duas.

- Opcionais – Passa a haver duas regras, uma contendo o token e outra não.

  RETURN
  RETURN Expr

- Repetição (incluindo 0): Criou-se uma nova regra auxiliar onde é declarado o caso base com recursividade à direita.

  StatementList Statement
  LBRACE StatementList RBRACE

Todas as regras excepto a regra “type” são do tipo “no”, que é um nó da AST. A regra “type” é do tipo “type_struct” que contém uma string com o nome do tipo.

2) Estrutura da AST e Tabela de Símbolos

AST

No desenvolvimento da AST, foi necessário criar uma estrutura auxiliar “node”, onde guarda o próprio nome, um ponteiro para o filho e para o nó seguinte. As funções “**create_node**”, “**add_next**”, “**add_son**”, “**is_block**”, “**new_id**”, “**print_tree**” e “**free_tree**” que executam as várias operações para criar, construir e imprimir a árvore.

- create_node – aloca memória e cria o nó;
- add_next – adiciona um próximo nó:
- add_son – adiciona um filho a um nó da árvore;
- is_block – verifica se o nó é ou não um bloco;
- new_id – cria um novo nó com um id específico;
- print_tree – imprime a árvore;
- free_tree – liberta a memória alocada pela árvore;

Logo estas funções servirão para criar a árvore AST aquando a análise sintática, com a exceção da print_tree e free_tree que irão ser utilizadas após a arvore ser construída.

Tabelas de Símbolos

Após a criação da arvore AST iremos proceder à criação das nossas tabelas de símbolos.

Iremos então percorrer a arvore duas vezes para a criação da tabela de símbolos. Na primeira passagem apenas iremos definir as funções/métodos e avaliar os seus headers, assim como as variáveis globais.

Já na próxima vez que percorrermos a árvore iremos entrar no body das funções onde serão analisadas as declarações de variáveis e os statements. Enquanto a análise de cada ramo da árvore é feita irão ser chamadas inúmeras funções check que terão a função de detetar os erros semânticos assim como imprimidos. Estas funções de check são ainda responsáveis pela anotação da nossa arvore, uma vez que são elas que irão atribuir tipos(int, double, etc) a cada nó da nossa AST.

pelo nó filho. As variáveis locais e respetivos tipos são registados na tabela.

Estruturas

É ainda importante referir as estruturas que dão vida à nossa tabela de símbolos, a estrutura table_t (Fig4) que contem o nome do nó da tabela, o seu id, um ponteiro para o primeiro e último simbolos e um ponteiro para o próximo simbolo, que será usado para percorrer a tabela. Para complementar criámos a estrutura symbol_t (Fig.3) onde guarda o nome do simbolo, o nó pai, o tipo, os parâmetros, a linha e a coluna, um ponteiro para percorrer a lista ligada e booleanos para saber se é função ou parâmetro e se já está a ser usada ou se já foi alocada.

```
typedef struct sym {
    char* name;
    type_t type;
    struct pt* paramtypes;
    int line, col;
    bool param;
    bool func;
    bool used;
    struct sym* next;
    int id;
} symbol_t;
```

Fig.3

```
typedef struct st {
    char* name;
    int classe;
    int id;
    struct sym* first;
    struct sym* last;
    struct st* next;
} table_t;
```

Fig.4

3) Geração de Código

Infelizmente, a nova situação em que vivemos trouxe um modelo de trabalho à distância do qual foi preciso os membros do grupo se adaptarem, tornando o processo produtivo mais lento. Apesar da implementação e bom funcionamento da tabela de símbolos e da AST em todos os casos de teste, a terceira meta não foi concluída, havendo ainda alguns erros semânticos por resolver. A última parte do projeto - geração de código – não foi desenvolvida a tempo da entrega.