Ficha prática 5

Compiladores

2019/20

Análise Semântica

1 Introdução

Com o Lex e o YACC, já fizemos exemplos dos passos de Análise Lexical e Análise Sintática. Passemos agora à Análise Semântica. Uma das tarefas desta última é criar uma tabela de símbolos, que seja posteriormente utilizável pelas fases seguintes. Nesta tabela, mais do que o valor dos símbolos propriamente dito, é importante guardar o seu nome e tipo. O nome e o tipo permitirão mais tarde fazer deteção de erros de vária ordem:

- Incompatibilidade entre tipos (*type mismatch*). Por exemplo, quando temos `a=b`, sendo `a` uma `string` e `b` um `double`;
- Redeclarações. Por exemplo, se estivermos a redeclarar uma variável já definida;
- Parâmetros de chamada de funções. Por exemplo, chamar a função `foo(1, 2, 3.2)` quando ela foi definida como `foo(double x, char* y, int z)`.

Note que estamos apenas a falar de variáveis. Dependendo da linguagem, existem muitos outros tipos de símbolos: funções, classes, constantes, operadores, namespaces, etc. Para simplificar, concentrar-nos-emos para já apenas em variáveis.

Nesta ficha, iremos concentrar-nos na criação e gestão da tabela de símbolos para a compilação.

Voltemos à linguagem de programação de cuja sintaxe se apresenta o seguinte exemplo:

```
let
    integer  n
    double   x
    char     z
```

```
in
    write n
    write x
end
```

Neste exemplo, e como estudado na ficha anterior, a construção da árvore de sintaxe abstracta (AST) é baseada num conjunto de funções `insert`. Podemos ver um extracto do ficheiro YACC:

```
...
program: LET vardeclist IN statementlist END
                                {$$=myprogram=insert_program($2, $4);}
    ;

vardeclist: /*empty*/           {$$=NULL;}
    | vardeclist vardec         {$$=insert_vardec_list($1, $2);}
    ;

vardec: INTEGER IDENTIFIER      {$$=insert_integer_dec($2);}
    | CHARACTER IDENTIFIER      {$$=insert_character_dec($2);}
    | DOUBLE IDENTIFIER         {$$=insert_double_dec($2);}
    ;

statementlist: /*empty*/        {$$=NULL;}
    | statementlist statement   {$$=insert_statement_list($1, $2);}
    ;

statement: WRITE IDENTIFIER     {$$=insert_write_statement($2);}
    ;
    ...
```

Após a análise sintática e construção da AST, é o momento de avançarmos para a análise semântica.

Esta análise basear-se-á na inspeção da AST, começando pela raíz (uma estrutura `is_program`). Para cada estrutura, teremos uma função `check_<nome da estrutura>` que se concentrará na verificação dessa estrutura. Tudo começa então em `main`:

```
...
int main(int argc, char **argv)
{
    int errors;
```

```
    yyparse();
    errors=check_program(myprogram);
        //CHAMADA À FUNÇÃO DE VERIFICAÇÃO DE is_program

    if(errors>0)
        printf("The program has %d errors!\n", errors);

    show table();
    return 0;
}
...
```

Por sua vez, a função `check_program` chamará as funções `check` de cada uma das suas componentes (no ficheiro `semantics.c`):

```
...
int check_program(is program* p)
{
    int errorcount=0;
    errorcount=check_vardec_list(p->vlist);
    errorcount+=check_statement_list(p->slist);
    return errorcount;
}
...
```

E assim sucessivamente:

```
...
int check_vardec_list(is_vardec_list* ivl)
{
    int errorcount=0;
    is_vardec_list* tmp;
    for(tmp=ivl; tmp; tmp=tmp->next)
        errorcount+=check_vardec(tmp->val);
    return errorcount;
}
...
```

Pretende-se então verificar a correção das variáveis. Para tal, teremos que construir uma tabela de símbolos (que conterá, para já, apenas variáveis). Estas poderão ser do tipo `integer`, `character` e `doub`.

Abaixo, vemos as estruturas necessárias para esta tabela de símbolos (que será uma lista ligada de estruturas `table_element`, definida em `symbol_table.h`).

```
typedef enum {integer, character, doub} basic_type;

typedef struct _t1{
    char name[32];
    basic_type type;
    struct _t1 *next;
} table_element;
```

Note-se que esta tabela de símbolos será representada por uma variável global (chamada `symtab` no nosso exemplo).

Cada `symbol_entry` terá então um nome e um tipo (`integer`, `character` e `doub`). À medida que vai encontrando declarações de variáveis, o nosso compilador deve então inserir o novo símbolo usando a função `insert_el` (definida no ficheiro `symbol_table.c`). Repare-se que também verifica se a variável já foi declarada:

```
...
int check_integer_dec(is_integer dec* iid)
{
    table_element* newel=insert_el(iid->id, integer);
    if(newel==NULL)
    {
        printf("Symbol %s already defined!\n", iid->id);
        return 1;
    }
    return 0;
}
...
```

Mais tarde, na verificação das statements de `write`, é importante verificar se o símbolo foi definido:

```
...
int check_write_statement(is_write_statement* iws)
{
    table_element* aux=search_el(iws->id);
    if(aux==NULL)
    {
```

```
        printf("Symbol %s not declared!\n", iws->id);
        return 1;
    }
    return 0;
}
...
```

Como terá notado, é feito o uso das funções `insert_el(char*, basictype)` e `search_el(char *str)`, que estão no ficheiro `symbol_table.c`:

```
//Insere um novo identificador na cauda de uma lista ligada de símbolos
table_element *insert_el(char *str, basic_type t)
{
    table_element *newSymbol=(table_element*) malloc(sizeof(table_element));
    table_element *aux;
    table_element* previous;

    strcpy(newSymbol->name, str);
    newSymbol->type=t;
    newSymbol->next=NULL;

    if(symtab) //Se table já tem elementos
    { //Procura cauda da lista e verifica se símbolo ja existe
      //(NOTA: assume-se uma tabela de símbolos globais!)
        for(aux=symtab; aux; previous=aux, aux=aux->next)
             if(strcmp(aux->name, str)==0)
                 return NULL;

        previous->next=newSymbol; //adiciona ao final da lista
    }
    else //symtab tem um elemento -> o novo símbolo
          symtab=newSymbol;
    return newSymbol;
}
...
//Procura um identificador, devolve 0 caso nao exista
table_element *search_el(char *str)
{
    table_element *aux;

    for(aux=symtab; aux; aux=aux->next)
          if(strcmp(aux->name, str)==0)
```

```
            return aux;
    return NULL;
}
```

Construa o analisador semântico a partir dos ficheiros `ficha5.exemplo.l`, `ficha5.exemplo.y`, `functions.c`, `semantics.c` e `symbol_table.c` disponibilizados com esta ficha.

Depois, execute o analisador com o ficheiro `test.s`. O resultado deverá ser:

```
symbol n, type 0
symbol x, type 2
symbol z, type 1
```

O output produzido é um *report* que é executado depois de `yyparse`, apenas para mostrar as variáveis que estão em `table`. Ou seja, neste momento o nosso analisador apenas guarda variáveis na tabela da símbolos.

Neste pequeno programa, é importante que repare em alguns aspectos:

- Trata-se de uma versão muito simples de análise semântica (repare que apenas se verifica se uma variável já foi declarada);
- Todas as variáveis são globais;
- Pretende-se sobretudo que se aperceba do mecanismo de criação e actualização de uma tabela de símbolos (que é implementada com uma lista ligada) e de uma forma simples de fazer verificações.

Para fazer uma análise semântica mais abrangente, terá que considerar a noção de *ambiente*, tal como ensinado nas aulas teóricas. Um ambiente corresponde a um conjunto de identificadores conhecidos num dado contexto. Por exemplo, uma função com parâmetros e variáveis locais terá no seu ambiente estes parâmetros e variáveis locais, bem como as variáveis globais do programa onde se encontra essa função.

2 Exercícios

Exercício 1 *Altere o programa de forma a mostrar a linha e a coluna onde ocorrem os erros.*

□

Exercício 2 *Substitua a primeira regra da sua gramática pela seguinte:*

```
program: program LET vardeclist IN statement_list END
       | LET vardeclist IN statement_list END
;
```

Com esta nova regra, o seu programa passará a conter vários sub-programas da linguagem. Cada sub-programa conterá uma lista de declarações de variáveis e uma lista de statements. Assuma que cada sub-programa não partilha as suas declarações com os outros sub-programas.

- *Faça as modificações necessárias para contemplar esta alteração.*

  ***Importante**: no final, o seu programa deverá listar as tabelas de símbolos (ambientes) de cada um dos sub-programas.*

□

Exercício 3 *A tabela de símbolos limita o comprimento dos identificadores a 31 caracteres. Elimine esta limitação e liberte toda a memória alocada durante a execução do analisador antes da função* `main` *retornar.*

□

Exercício 4 *Acrescente a esta linguagem a possibilidade de inicializar as variáveis declaradas com valores de tipo adequado e de realizar operações aritméticas envolvendo essas variáveis e/ou valores constantes.*

□

Referências

[1] Anexo A de Processadores de Linguagens. Rui Gustavo Crespo. IST Press. 1998

[2] Manuais da linguagem C (por exemplo, Linguagem C, Luís Damas, FCA. 1999)

[3] Modern compiler implementation in C. A. Appel. Cambridge Press. 1998.