Trabalho Prático 3 - Analisador Sintático

Alunos:

- Lucas Caetano Lopes Rodrigues, matrícula: 2016006670
- Lucas Starling de Paula Salles, matrícula: 2016006697

Implementação do Analisador Sintático

Para implementação da etapa de análise léxica, utilizamos as ferramentas `flex` e `yacc`. A ferramenta `flex` é utilizada para a criação do analisador léxico da linguagem, como definido na última etapa deste Trabalho Prático. Após a geração da implementação do analisador léxico na linguagem C pela ferramenta `flex`, geramos o analisador sintático utilizando a ferramenta `yacc`, que recebe como entrada um arquivo que contém as regras de redução da gramática, seus *tokens*, ordem de precedência e tratamento de erros da etapa de análise sintática.

Os *tokens* da gramática foram definidos através das expressões regulares mostradas a seguir. Discutimos, em seguida, os detalhes de escolhas para a criação dos *tokens* são discutidos em seguida.

A gramática a seguir é definida no arquivo `sa-generator.y`. Ela contém as mesmas regras da gramática descrita na proposta do Trabalho Prático, exceto por pequenas alterações que foram necessárias para a correção de conflitos na gramática:

1. O não-terminal `fator_a` na regra

   ```
   fator_a := "-" factor
            | factor
   ```

   foi alterada para `factor_a` para manter a consistência na gramática.

2. Removemos a produção `identifier` da regra

   ```
   factor := identifier
           | constant
           | "(" expr ")"
           | function_ref
           | NOT factor
   ```

   visto que a regra de `function_ref` já possui um `identifier`.

A gramática foi, portanto, definida da seguinte forma:

```
// Arquivo sa-generator.y
%{
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "lex.yy.c"

void yyerror(const char *str)
```

```
{
        fprintf(stderr,"error: %s\n",str);
}

%}

%token PROGRAM
%token END_TOKEN THEN_TOKEN
%token CHAR_TYPE BOOL_TYPE REAL_TYPE INTEGER_TYPE ELSE_TOKEN
%token CHAR_CONST CONST  BOOL_CONST IDENTIFIER TRUE_CONST FALSE_CONST SIGN
%right OPEN_P BEGIN_TOKEN IF_TOKEN NOT_TOKEN WHILE_TOKEN UNTIL_TOKEN
%left CLOSE_P

%token UNKNOWN RELOP FUNC MULOP ADDOP
%token GOTO_TOKEN WRITE_TOKEN READ_TOKEN
%token DO_TOKEN SEMICOLON
%token COLON COMMA ASSIGN

%%

program:                    PROGRAM IDENTIFIER SEMICOLON decl_list compound_stmt
                            ;
decl_list:                  decl_list SEMICOLON decl
                            | decl
                            ;
decl:                       ident_list COLON type
                            ;
ident_list:                 ident_list COMMA IDENTIFIER
                            | IDENTIFIER
                            ;
type:                       INTEGER_TYPE
                            | REAL_TYPE
                            | BOOL_TYPE
                            | CHAR_TYPE
                            ;
compound_stmt:              BEGIN_TOKEN stmt_list END_TOKEN
                            ;
stmt_list:                  stmt_list SEMICOLON stmt
                            | stmt
                            ;
stmt:                       label COLON unlabelled_stmt
                            | unlabelled_stmt
                            ;
label:                      IDENTIFIER
                            ;
unlabelled_stmt:            assign_stmt
                            | if_stmt
                            | loop_stmt
                            | read_stmt
                            | write_stmt
                            | goto_stmt
                            | compound_stmt
                            ;
assign_stmt:                IDENTIFIER ASSIGN expr
                            ;
if_stmt:                    IF_TOKEN cond THEN_TOKEN stmt /* generates shift-
reduce, shift is default on yacc */
                            | IF_TOKEN cond THEN_TOKEN stmt ELSE_TOKEN stmt
```

```
                        ;
cond:                   expr
                        ;
loop_stmt:              stmt_prefix DO_TOKEN stmt_list stmt_suffix
                        ;
stmt_prefix:            WHILE_TOKEN cond
                        |
                        ;
stmt_suffix:            UNTIL_TOKEN cond
                        | END_TOKEN
                        ;
read_stmt:              READ_TOKEN OPEN_P ident_list CLOSE_P
                        ;
write_stmt:             WRITE_TOKEN OPEN_P expr_list CLOSE_P
                        ;
goto_stmt:              GOTO_TOKEN IDENTIFIER
                        ;
expr_list:              expr
                        | expr_list COMMA expr
                        ;
expr:                   simple_expr
                        | simple_expr RELOP simple_expr
                        ;
simple_expr:            term
                        | simple_expr ADDOP term
                        ;
term:                   factor_a
                        | term MULOP factor_a
                        ;
factor_a:               SIGN factor
                        | factor
                        ;
factor:                 constant /* removed IDENTIFIER (in function_ref
already) */
                        | OPEN_P expr CLOSE_P
                        | function_ref
                        | NOT_TOKEN factor
                        ;
function_ref:            IDENTIFIER
                        | function_ref_par
                        ;
function_ref_par:       variable OPEN_P expr_list CLOSE_P
                        ;
variable:               simple_variable_or_proc
                        | function_ref_par
                        ;
simple_variable_or_proc:    IDENTIFIER
                        ;
constant:               CONST
                        | CHAR_CONST
                        | boolean_constant
                        ;
boolean_constant:       FALSE_CONST
                        | TRUE_CONST
                        ;


%%
```

```
int main(int argc, char *argv[]) {
        if (argc == 1){
          yyparse();
        }

        if (argc == 2) {
          yyin = fopen(argv[1], "r");
          yyparse();
        }

        return 0;
}
```

Como pode-se ver pelo código acima, a regra `if_stmt` gera um conflito *shift_reduce*. Tentamos algumas maneiras de resolver esse problema, conhecido como *the dangling 'else' problem*. Optamos por deixar com que o `yacc` resolva o conflito (por padrão, o `yacc` resolve o conflito escolhendo realizar o *shift*).

Geração do Analisador Sintático e Testes

Para gerar o programa que faz a análise sintática, primeiro precisamos gerar o analisador léxico utilizando a ferramenta `flex`. Em seguida, geramos o analisador sintático utilizando a ferramenta `yacc`.

```
$ flex Ylex.lex
$ yacc sa-generator.y -d
```

Os seguintes arquivos são gerados: `lex.yy.c`, `y.tab.c`, `y.tab.h`. Agora, basta compilar o arquivo `y.tab.c` utilizando o `gcc`:

```
$ cc y.tab.c -ll
```

Um executável `a.out` será gerado no diretório.

Alternativamente, pode-se gerar o executável `a.out` a partir do *script* `compile.sh`, na raíz do projeto:

```
$ ./compile.sh
```

Realizando testes

Para a realização de testes, criamos alguns programas na linguagem P na pasta `programs`, que são utilizados como entrada para o analisador sintático gerado:

```
// error
program error;
it1,it2,it3,it4: integer;
fl1: real
begin
it1 := 5;
it2 := 3;
it3 := 4;
it4 := 2;
if it1 == it2
then
fl1 := (it1 * it2) / (it3 * it4)
end
```

```
// error-fixed
program errorfixed;
it1,it2,it3,it4: integer;
fl1: real
begin
it1 := 5;
it2 := 3;
it3 := 4;
it4 := 2;
if it1 = it2
then
fl1 := (it1 * it2) / (it3 * it4)
end
```

```
program one;
id1: integer
begin
id1 := 5
end
```

```
program three;
it1,it2,it3,it4,it5: integer
begin
it1 := 5;
it2 := 3;
it3 := 4;
it4 := 2;
it5 := (it1 * it2) / (it3 * it4)
end
```

Para testá-los, basta chamar o executável `a.out` passando cada programa como parâmetro:

```
$ ./a.out programs/error
error: syntax error
$ ./a.out programs/error-fixed
$ ./a.out programs/one
...
```

Conclusão

As ferramentas `flex` e `yacc` são enormes facilitadores para a geração dos analisadores léxico e sintático. Em conjunto, as ferramentas definem um arcabouço completo para o desenvolvimento do *front end* de compiladores.

Após pequenas modificações na gramática, foi possível gerar ambos os analisadores léxico e sintático com poucas linhas de código, e com isso o *front end* do compilador está completo.