Universidade Federal de Santa Catarina
Departamento de Informática e Estatística
INE5426 - Construção de Compiladores

Analisador Léxico

Alunos:
Caetano Colin Torres
Alek Frohlich
Nícolas Goeldner

Florianópolis, 2021

Sumário

1. Identificação dos Tokens

Na análise léxica é necessário converter determinadas sequências de caracteres em Tokens. Ao todo, no nosso trabalho, temos 33 possíveis Tokens gerados. Ao compilar o programa no arquivo ***xpp.tab.c*** pode-se ver os números associados a cada Token conforme a Figura 1. Os Tokens que possuem apenas um caractere são identificados pelo seu valor em ASCII. Abaixo vemos listados todos os possíveis Tokens:

Token **DEF**
Token **INT**
Token **FLOAT**
Token **STRING**
Token **BREAK**
Token **PRINT**
Token **READ**
Token **RETURN**
Token **IF**
Token **ELSE**
Token **FOR**
Token **NEW**
Token **NUL**
Token **CMP**
Token **IDENT**
Token **STRING_C**
Token **INT_C**
Token **FLOAT_C**
Token **(**
Token **)**
Token **{**
Token **}**
Token **[**
Token **]**
Token **(**
Token **;**
Token **,**
Token **=**
Token **+**
Token **-**
Token *****
Token **/**
Token **%**

```
/* Token type.  */
#ifndef YYTOKENTYPE
# define YYTOKENTYPE
  enum yytokentype
  {
    DEF = 258,
    INT = 259,
    FLOAT = 260,
    STRING = 261,
    BREAK = 262,
    PRINT = 263,
    READ = 264,
    RETURN = 265,
    IF = 266,
    ELSE = 267,
    FOR = 268,
    NEW = 269,
    NUL = 270,
    CMP = 271,
    OP = 272,
    IDENT = 273,
    STRING_C = 274,
    INT_C = 275,
    FLOAT_C = 276
  };
#endif
```

Figura 1. Identificação dos Tokens Gerados em uma Enumeração.

2. Produção das Definições Regulares para Tokens

As definições regulares para cada token são definidas no arquivo ***xpp.l***. A maioria dos tokens são literais, ou seja tem sempre o mesmo valor e são definidos de forma mais simples. Exemplos de tokens literais são as keywords na Figura 2. Exemplos de tokens não literais estão na Figura 3, onde temos expressões regulares mais complexas como forma de identificar cadeias de caracteres que se encaixam em tokens.

```
    /* Keywords */
def                 { return DEF;       }
int                 { return INT;       }
float               { return FLOAT;     }
string              { return STRING;    }
break               { return BREAK;     }
print               { return PRINT;     }
read                { return READ;      }
return              { return RETURN;    }
if                  { return IF;        }
else                { return ELSE;      }
for                 { return FOR;       }
new                 { return NEW;       }
null                { return NUL;       }
```

Figura 2. Tokens literais para Keywords.

```
white               [ \t]+
digit               [0-9]
 /* TODO: improve ident pattern */
ident               [a-zA-Z_][a-zA-Z0-9_]*
int_constant        {digit}+
float_constant      {int_constant}(\.{int_constant})?
 /* Based of C-style strings; c.f. https://www.lysator.liu.se/c/ANSI-C-grammar-l.html */
string_constant     \"([^\"\\]|\\.)*\"
```

Figura 3. Definiçao de tokens não literais.

```
    /* Punctuation */
"("                 { return '('; }
")"                 { return ')'; }
"{"                 { return '{';  }
"}"                 { return '}';  }
"["                 { return '[';  }
"]"                 { return ']';  }
";"                 { return ';';  }
","                 { return ',';  }
"="                 { return '=';  }
```

Figura 4. Definição de tokens para pontuação.

```
    /* Operators */
"<"                 { return CMP;   }
">"                 { return CMP;   }
"<="                { return CMP;   }
">="                { return CMP;   }
"=="                { return CMP;   }
"!="                { return CMP;   }
"+"                 { return '+';   }
"-"                 { return '-';   }
"*"                 { return '*';   }
"/"                 { return '/';   }
"%"                 { return '%';   }
```

Figura 5. Definição de Tokens de operadores.

```
    /* Named rules */
{white}             {}
{ident}             { yylval.sval   = strdup(yytext); return IDENT; }
{string_constant}   {
                      char * dup = strdup(yytext);
                      dup++;
                      dup[strlen(dup)-1] = 0;
                      yylval.sval   = dup;
                      return STRING_C;
                    }
{int_constant}      { yylval.ival    = atoi(yytext); return INT_C; }
{float_constant}    { yylval.fval   = atof(yytext); return FLOAT_C; }
```

Figura 6. Regras aplicadas para certos Tokens.

3. Diagramas de Transição

Os diagramas de transição são representações visuais que auxiliam na visualização e modelagem das expressões que definem os Tokens, cada token irá possuir seu próprio diagrama que identifica a cadeia de caracteres ao qual aquele token pertence, tokens literais terão diagramas mais simples e diretos e tokens não literais terão diagramas com uma lógica mais elaborada. Usamos expressões regulares em alguns diagramas para simplificar, por exemplo o Diagrama 16, que aponta ocorrências de Tokens em sequências de caracteres que começam com caracteres de “a” até “z” minúsculos ou maiúsculos ou “_” seguidos de caracteres de “a” até “z” minúsculos ou maiúsculos ou “_” ou digitos de “0” até “9”. O Diagrama 19 reconhece sequências que começam com aspas duplo e terminam com aspas duplo, o conteúdo dentro da aspas duplo pode ser qualquer caractere que não seja aspas duplo ou barra inversa ou que seja barra inversa seguido de algum caracter.

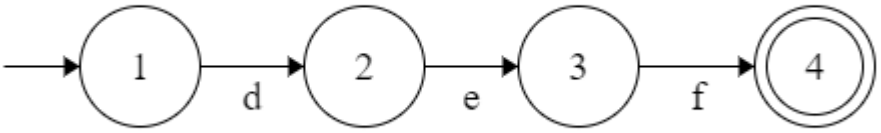

Diagrama 1. **def** keyword.

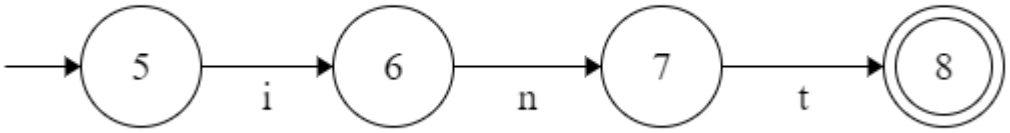

Diagrama 2. **int** keyword.

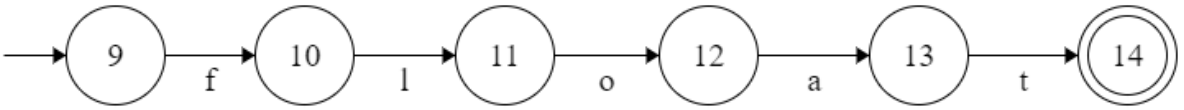

Diagrama 3. **float** keyword.

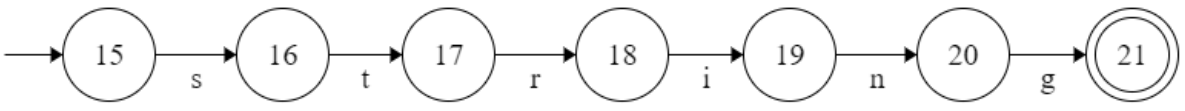

Diagrama 4. **string** keyword.

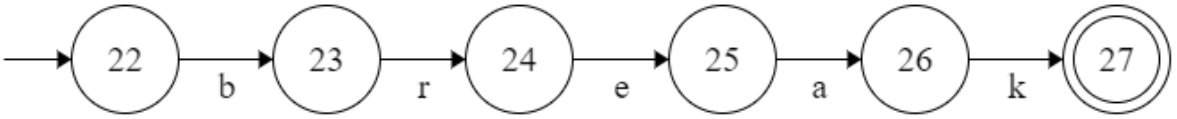

Diagrama 5. **break** keyword.

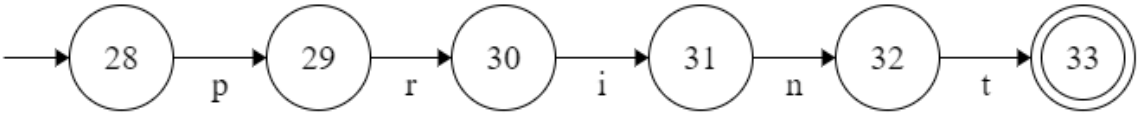

Diagrama 6. **print** keyword.

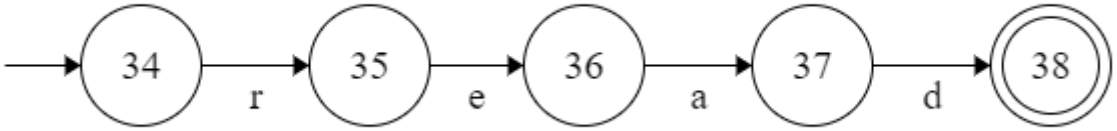

Diagrama 7. **read** keyword.

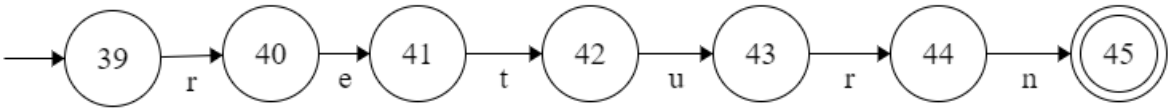

Diagrama 8. **return** keyword.

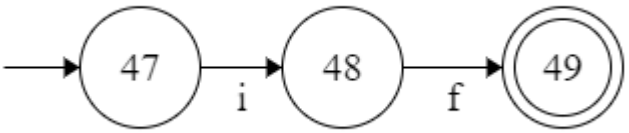

Diagrama 9. **if** keyword.

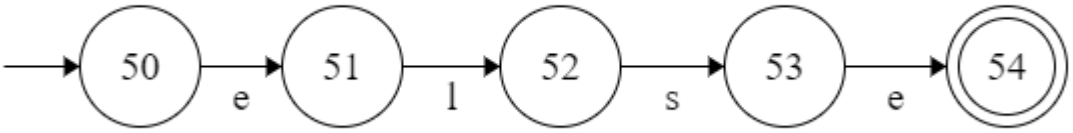

Diagrama 10. **else** keyword**.**

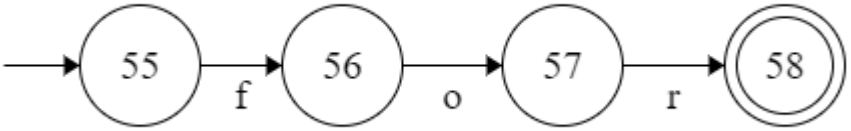

Diagrama 11. **for** keyword.

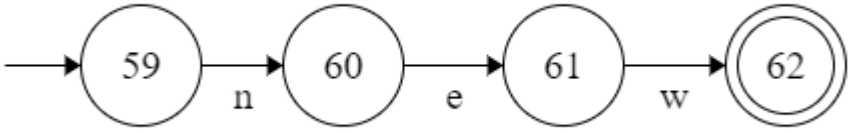

Diagrama 12. **new** keyword.

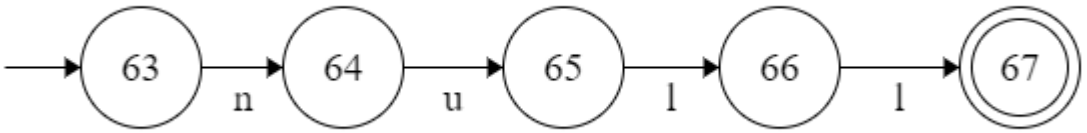

Diagrama 13. **null** keyword.

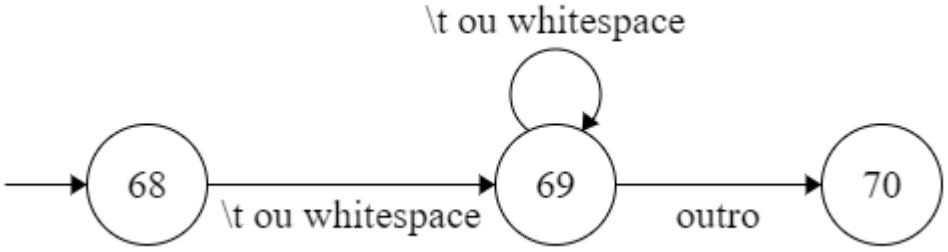

Diagrama 14. definição regular para **white.**

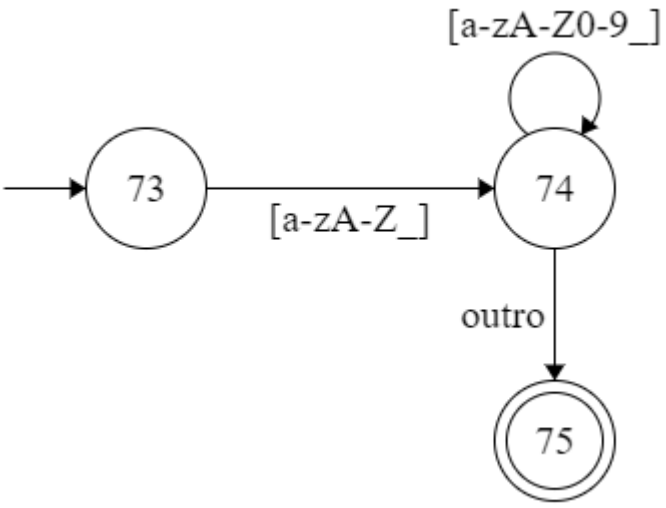

Diagrama 15. definição regular para **ident.**

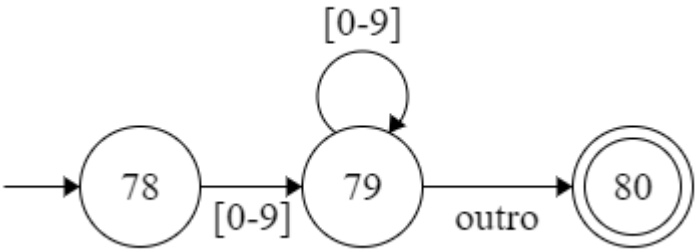

Diagrama 16. definição regular para **int_constant.**

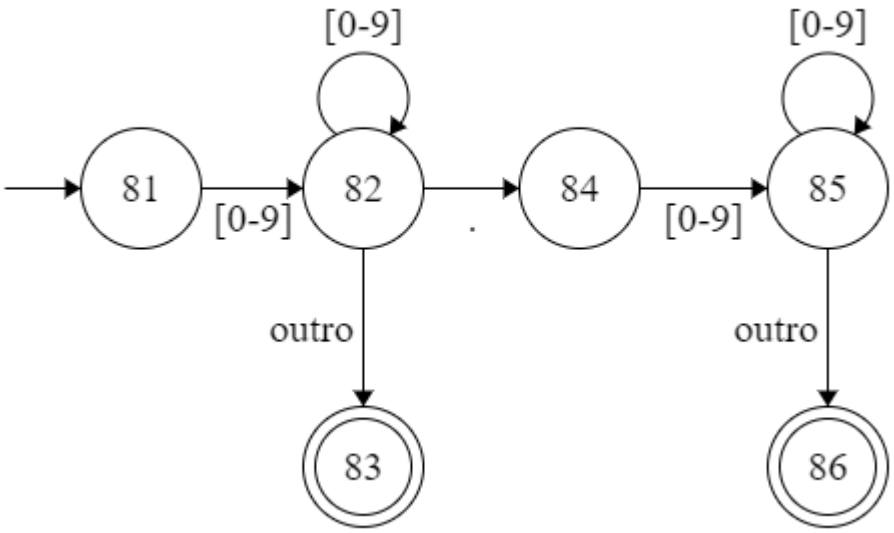

Diagrama 17. definição regular para **float_constant.**

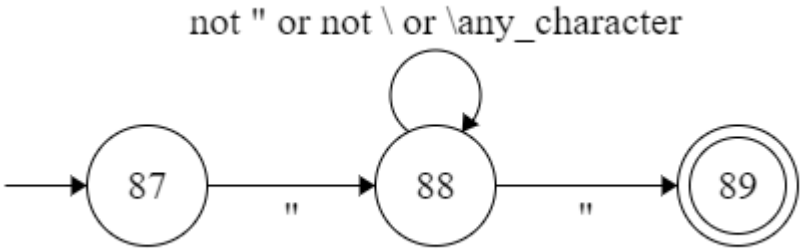

Diagrama 18. definição regular para **string_constant.**

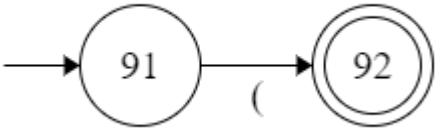

Diagrama 19. definição regular para pontuação "**(**"**.**

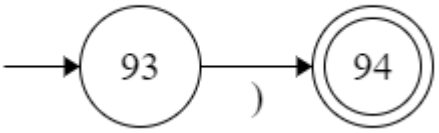

Diagrama 20. definição regular para pontuação "**)**"**.**

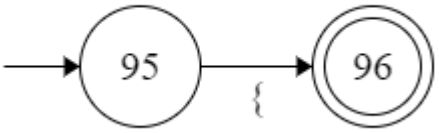

Diagrama 21. definição regular para pontuação "**{**".

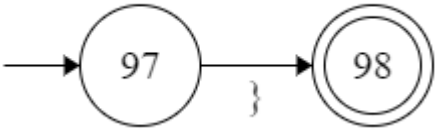

Diagrama 22. definição regular para pontuação "**}**".

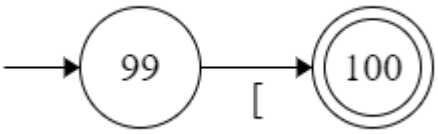

Diagrama 23. definição regular para pontuação "**[**".

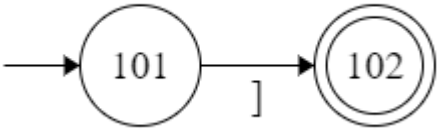

Diagrama 24. definição regular para pontuação "**]**".

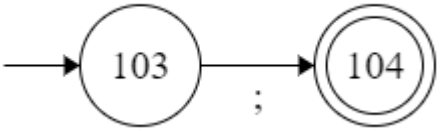

Diagrama 25. definição regular para pontuação "**;**".

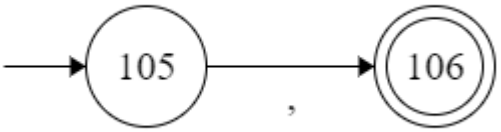

Diagrama 26. definição regular para pontuação "**,**".

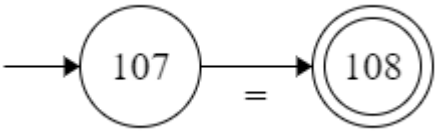

Diagrama 27. definição regular para pontuação "**=**".

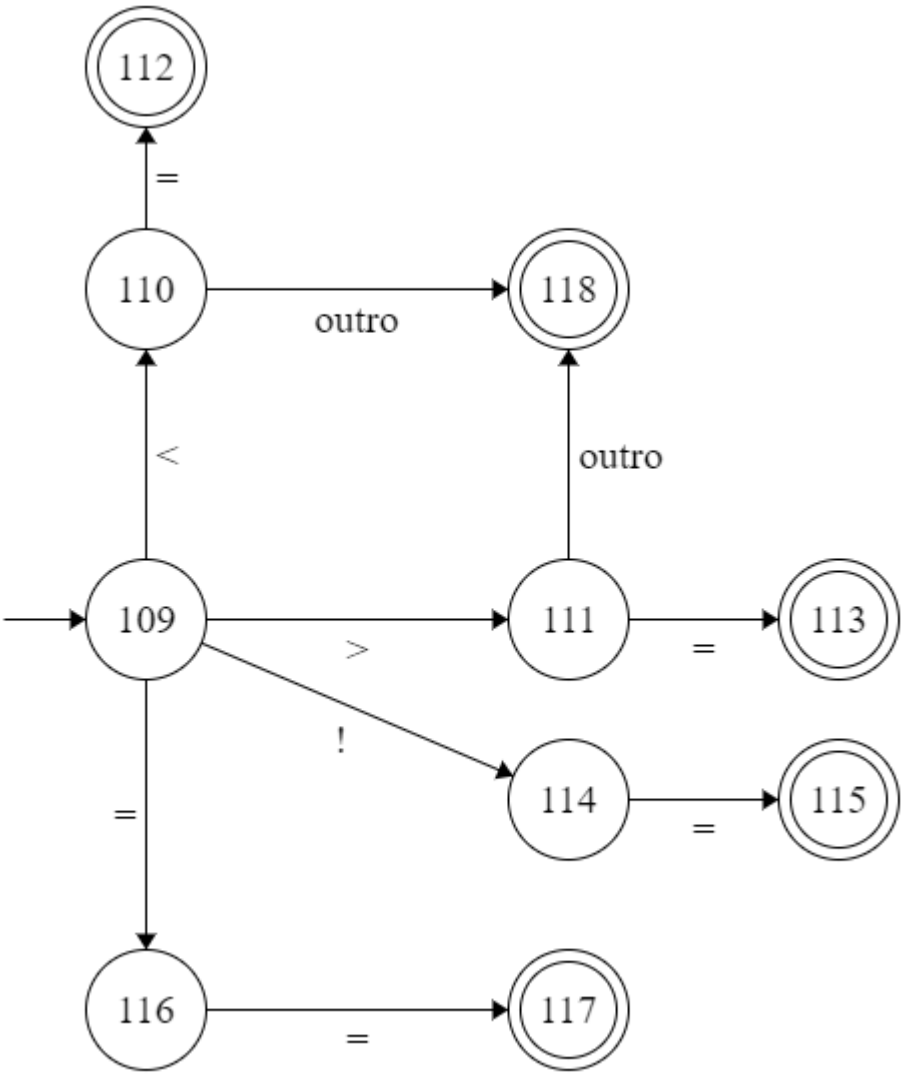

Diagrama 28. definição regular para token CMP.

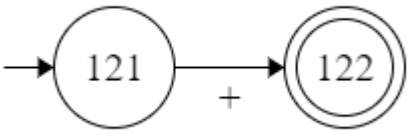

Diagrama 29. definição regular para operador “+“**.**

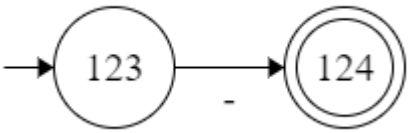

Diagrama 30. definição regular para operador "**-**"**.**

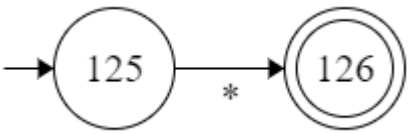

Diagrama 31. definição regular para operador "*****"**.**

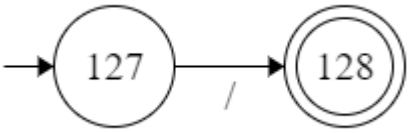

Diagrama 32. definição regular para operador "/"**.**

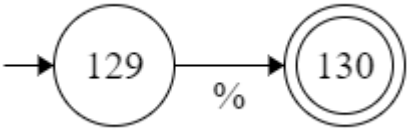

Diagrama 33. definição regular para operador "**%**"**.**

4. Tabela de Símbolos

A Tabela de símbolos é uma estrutura de dados utilizada para o armazenamento de informações de identificadores, onde cada identificador é associado com uma informação relacionada a declaração ou ocorrência no código fonte - como tipo e layout de armazenamento. Basicamente a tabela armazena informações relacionadas a símbolos de entrada. No caso do nosso trabalho, a tabela de símbolos foi implementada na fase de análise sintática.

Na nossa tabela de símbolos utilizamos uma stack cujos elementos são os vários escopos do código. Assim que abrimos um escopo, ele é colocado na stack e, quando o fechamos, damos pop. A ideia é que conforme vamos entrando em escopos mais profundos, vamos aumentando a stack, mantendo os escopos pai no fundo até que eles sejam fechados, com isso podemos obter informações de escopos pai. Por enquanto, estamos armazenando apenas o tipo do identificador, podendo este ser: INT, FLOAT, STRING, FUNCTION. Os símbolos armazenados são funções e variáveis.

Ilustramos esse processo de aninhamento de escopos com auxílio da figura 7, que contém o trecho da gramática responsável pela definição de funções. Começamos a declaração de uma função, colocando seu nome no escopo global e então abrimos o escopo de seus parâmetros; cada parâmetro é colocado na tabela; depois abrimos o escopo para o corpo da função, ou seja, o que está entre as chaves sem fechar o escopo dos parâmetros; depois que termina a função (fecha as chaves), fechamos o escopo do corpo e depois fechamos o escopo dos parâmetros, terminando a definição função.

```
funclist: funcdef funclist | funcdef;
funcdef: DEF IDENT                    { if(!check_put(std::string($2), SymType::T_FUNC)) YYABORT; Env::open_scope();}
                '(' paramlist ')' '{' { Env::open_scope(); }
                    statelist '}'     { Env::close_scope(); Env::close_scope(); }
;

paramlist: type IDENT { if(!check_put(std::string($2), $1)) YYABORT; } ',' paramlist
        |  type IDENT { if(!check_put(std::string($2), $1)) YYABORT; }
        |  %empty
;
type: INT { $$ = 0; } | FLOAT { $$ = 1; } | STRING { $$ = 2; };
```

Figura 7. Exemplo de resolução de escopo.

5. Descrição da Ferramenta

Usamos a ferramenta ***flex*** para gerar um analisador léxico, e posteriormente usamos o Bison para gerar um analisador sintático LALR. ***flex*** lê de uma stream de entrada especificando o formato do analisador léxico (.l) contendo pares de definições regulares e fragmentos de código (*rules*), e gera como saída um arquivo fonte na linguagem C que define uma rotina ***yylex()***. Esse arquivo é compilado e ligado a arquivos .cpp utilizando ***g++*** para produzir um executável. Quando o executável é rodado, ele analisa a entrada para ocorrências das expressões regulares e executa ações (trechos de código) relacionadas a cada padrão, ao passo que este é encontrado.

Um exemplo de como usar ***flex*** é o input a seguir que especifica um scanner que quando encontra a string ‘username’ substitui pelo nome do usuário logado.

```
%%
username    printf( "%s", getlogin() );
```

Por padrão, qualquer texto que não tiver um ***match*** pelo scanner ***flex*** é copiado para o output. Assim, o efeito do scanner é copiar o arquivo entrada para a saída com cada ocorrência de 'username' expandida. No exemplo acima, só existe uma regra: 'username' é o padrão e 'printf' é a ação. O símbolo %% marca o início das regras.

Um arquivo de entrada ***flex*** consiste de três seções separadas por uma linha contendo ***%%***.

```
definitions
%%
rules
%%
user code
```

A seção de definições contém declarações de nome simples para simplificar a especificação do analisador. Definições de nome tem o formato ***name definition***, Um exemplo é:

```
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
```

Que define 'DIGIT' como uma expressão regular que marca ocorrências de um dígito único, e 'ID' é a expressão regular que marca ocorrências de uma letra seguida de um ou mais letras/dígitos.

A seção de regras contém uma série de regras na forma ***pattern action***; pode-se definir variáveis nessa seção que são locais para a rotina do analisador. Na seção de user code, o código é apenas copiado para ***lex.yy.c*** e pode ser usado para rotinas que são chamadas ou chamam o analisador.

A saída do ***flex*** é o arquivo ***lex.yy.c***, que contém a rotina de análise léxica ***yylex()***, um número de tabelas usadas pelos tokens e um número de rotinas e macros auxiliares. Por padrão, ***yylex()*** é declarado como:

```
int yylex()
    {
    ... various definitions and the actions in here ...
    }
```

Toda vez que ***yylex()*** é chamada, a entrada é examinada até que outra ocorrência de um dos padrões seja encontrada. Quando um padrão é reconhecido, ***yylex()*** retorna um identificador avisando qual token foi encontrado. Esse identificador pode ser qualquer inteiro, porém, como utilizamos o *flex* em conjunto com o *Bison* (gerador de analisador sintático), seguimos a convenção em que os identificadores vem de uma enumeração (**yytokentype**); esta enumeracáo é definida pelo arquivo **xpp.tab.h**, que é gerado pela ferramenta Bison, conforme Figura 1 vista na seção 1. O valor do token é armazenado na global **yylval**.