

Analizador Léxico e Sintático

Vitor Fernandes Dullens - 16/0148260

Universidade de Brasília

1 Introdução

Este relatório abordará sobre as primeiras etapas do processo de criação de um tradutor, que consiste na implementação do analisador léxico e sintático para uma linguagem específica, como apresentado no livro base da disciplina [ALSU07]. Neste documento serão descritos com detalhes a motivação por trás desta implementação, assim como a descrição das análises e instruções para a compilação e execução do programa. Também será apresentada uma descrição da linguagem a ser analisada.

2 Motivação

Na linguagem C, muitas vezes sentimos falta de operações que facilitam a utilização de conjuntos, como existem em outras linguagens de mais alto nível. Dito isso, a fim de facilitar essas operações, uma implementação de uma nova primitiva de dados para conjuntos foi proposta dentro da linguagem C – **set**, assim como operações para a mesma – **add**, **remove**, entre outras.

Abaixo segue um exemplo de código na nova linguagem proposta.

```
int main() {
    set s;
    s = EMPTY;

    add(1 in add(2 in add(3 in s)));
    /* s = (1, 2, 3) */

    remove(1 in s);
    /* s = (2, 3) */
}
```

Além de operações com conjuntos, também foi adicionado um tipo polimórfico – **elem** que facilita, também, o uso de conjuntos. Mais detalhes sobre a linguagem podem ser encontrados no apêndice A.

3 Descrição da análise léxica

Para a implementação do analisador léxico foi utilizado o programa *Fast Lexical Analyzer Generator* - FLEX [Est17], que consiste em uma ferramenta geradora de programas que reconhecem padrões léxicos em textos.

No arquivo de nome `lexico.1` é possível visualizar as regras léxicas. Para o tratamento das mesmas são declaradas expressões regulares (regex) que as identificam e após essas declarações existe uma sequência de ações que o analisador executa ao encontrar uma regra. Além das regras e ações, no arquivo `lexico.1` também foram definidas duas variáveis `int linha`, `int coluna`, que representam, respectivamente, a linha e a coluna na qual está acontecendo a ação, e uma variável externa `int erros` que abriga a quantidade de erros que foram encontrados durante toda a execução das análises.

O analisador léxico também é responsável pela atribuição dos tokens que serão utilizados durante a análise sintática. Os tokens são declarados como uma `struct Lexema` que possui os parâmetros `char corpo[100]` que consiste no token propriamente dito, `int linha` e `int coluna` que, respectivamente, representam a linha e coluna daquele token. Essa `struct` é declarada no arquivo do analisador sintático, e pode ser utilizada no analisador léxico utilizando a variável global `yylval`.

4 Descrição da análise sintática

Para a implementação do analisador sintático foi utilizado o programa Bison [DS21], que consiste em um gerador de parser que utiliza de uma gramática livre do contexto para criar uma derivação LR. Para este trabalho foi utilizada a flag `%define lr.type canonical-lr` para que a derivação realizada seja a LR(1) canônica.

No arquivo de nome `sintatico.y` é possível visualizar a gramática presente no apêndice A com algumas modificações para se encaixar na sintaxe do próprio Bison.

4.1 Árvore Sintática

Pelo modo como é feito o parser pelo próprio Bison, é possível construir uma árvore sintática abstrata. Para isso, cada não terminal agora é um nó da árvore, e cada terminal é um símbolo, assim como será apresentado na Seção 4.2. Cada um desses nós, consiste em uma estrutura que armazena o símbolo (terminal), qual a regra que foi vista, o nó próximo e o filho. Com essas informações é possível realizar um caminho em profundidade a partir da raiz apresentando cada um dos nós que foram visitados e todos os detalhes necessários.

A Figura 1 representa a árvore gerada para um programa de entrada simples.

4.2 Tabela de Símbolos

Durante a passagem do analisador sintático, ele também é responsável por salvar os símbolos, afim de utiliza-los futuramente na análise semântica.

A tabela de símbolos é em uma lista de símbolos, onde um símbolo consiste em uma estrutura que armazena dados que podem ser úteis na próxima etapa do projeto, como o ID da variável ou função declarada, suas respectivas linha e coluna, tipo, se é uma função ou apenas variável e o escopo.

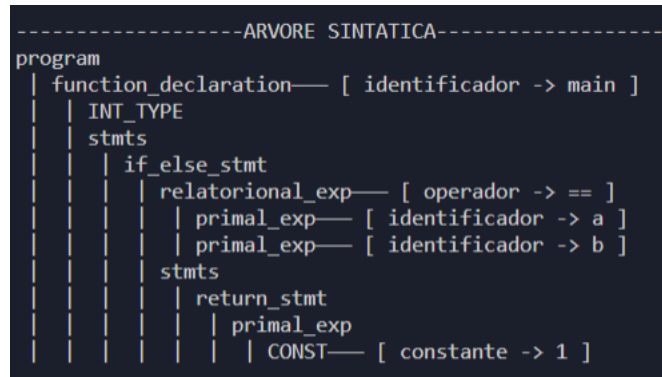


Figura 1. Árvore Sintática para uma entrada simples

TABELA DE SIMBOLOS				
IDENTIFICADOR	LINHA:COLUNA	TIPO	EH FUNCAO?	ESCOPO
f	1:5	INT	1	1
y	1:11	INT	0	1
s	2:9	SET	0	1
el	4:10	ELEM	0	1
x	5:9	INT	0	1
outrafuncao	9:5	INT	1	2
c	9:35	INT	0	2
b	9:28	INT	0	2
a	9:21	INT	0	2
main	14:5	INT	1	4
a	15:11	FLOAT	0	4

Acima é possível visualizar a tabela de símbolos para o arquivo `sucesso1.c`.

5 Descrição dos arquivos testes

Os testes se encontram na pasta `testes/`, os arquivos `sucesso1.c` e `sucesso2.c`, são testes que contém código correto, já os arquivos `erro1.c` e `erro2.c` contém códigos incorretos, sendo os seus erros, respectivamente:

1. SYNTAX ERROR 2:13 - syntax error, unexpected ';'
 - LEXICAL ERROR 3:20 - Unidentified character: #
 - SYNTAX ERROR 4:9 - syntax error, unexpected INTEGER, expecting ID
 - SYNTAX ERROR 9:5 - syntax error, unexpected INT_TYPE, expecting ';'

- ```
2. SYNTAX ERROR 5:21 - syntax error, unexpected ';',
 expecting ',', ' or ')'
SYNTAX ERROR 12:11 - syntax error, unexpected ';'

```

Basicamente, como é possível observar acima, os erros são apresentados e possuem a linha e coluna na qual eles pertencem (linha:coluna). O primeiro arquivo de erro - `erro1.c` - possui tanto erros léxicos, quanto erros sintáticos, já o segundo arquivo - `erro2.c` - possui apenas erros sintáticos.

Quando um arquivo possui erros não é apresentada a árvore sintática, apenas os erros e a tabela de símbolos gerada (que pode não estar completa dependendo do tipo dos erros). Além disso, cada erro é apresentado seguido de uma descrição do mesmo.

## 6 Instruções para compilação e execução do programa

O programa foi criado e testado em um sistema operacional Linux - Ubuntu 20.04.1 LTS. É necessária a instalação do FLEX [Est17] e do BISON [DS21] para a compilação do programa. Ao executar o programa também deverá ser passado o arquivo que será analisado.

Comandos para compilação e execução:

```
$ bison sintatico.y
$ flex lexico.l
$ gcc sintatico.tab.c lex.yy.c tabela.c arvore.c
$./a.out testes/<nome-arquivo>.c

```

Outra alternativa para facilitar a compilação seria utilizar o comando `make`.

Abaixo segue as versões de cada software que foi utilizado para o desenvolvimento:

```
$ gcc --version
gcc (Ubuntu 9.3.0-17ubuntu1~20.04) 9.3.0
$ bison --version
bison (GNU Bison) 3.7.6
$ flex --version
flex 2.6.4

```

## Referências

- [ALSU07] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, & Tools*. Pearson/Addison Wesley, 2nd edition, 2007.
- [DS21] C. Donnelly and R. Stallman. Bison - the yacc-compatible parser generator, 2021. Online; Acessado 18 de março de 2021  
<https://www.gnu.org/software/bison/manual/bison.html>.
- [Est17] W. Estes. Flex: Fast lexical analyzer generator, 2017. Online; Acessado 21 de fevereiro de 2021  
<https://github.com/westes/flex>.

## A Gramática

$\langle \text{program} \rangle ::= \langle \text{declaration\_list} \rangle$   
 $\langle \text{declaration\_list} \rangle ::= \langle \text{declaration} \rangle \langle \text{declaration\_list} \rangle \mid \langle \text{declaration} \rangle$   
 $\langle \text{declaration} \rangle ::= \langle \text{function\_declaration} \rangle \mid \langle \text{var\_declaration} \rangle$   
 $\langle \text{var\_declaration} \rangle ::= \langle \text{type} \rangle \langle \text{id} \rangle \text{' ;'}$   
 $\langle \text{function\_declaration} \rangle ::= \langle \text{type} \rangle \langle \text{id} \rangle \text{' (' } \langle \text{params\_list} \rangle \text{' )' } \langle \text{brackets\_stmt} \rangle$   
 $\quad \mid \langle \text{type} \rangle \langle \text{id} \rangle \text{' (' ' ' } \langle \text{brackets\_stmt} \rangle$   
 $\langle \text{params\_list} \rangle ::= \langle \text{type} \rangle \langle \text{id} \rangle \text{' , ' } \langle \text{param\_list} \rangle \mid \langle \text{type} \rangle \langle \text{id} \rangle$   
 $\langle \text{stmts} \rangle ::= \langle \text{stmt} \rangle \langle \text{stmts} \rangle \mid \langle \text{stmt} \rangle$   
 $\langle \text{stmt} \rangle ::= \langle \text{for\_stmt} \rangle \mid \langle \text{if\_else\_stmt} \rangle \mid \langle \text{return\_stmt} \rangle \mid \langle \text{io\_stmt} \rangle$   
 $\quad \mid \langle \text{brackets\_stmt} \rangle$   
 $\quad \mid \langle \text{exp\_stmt} \rangle$   
 $\quad \mid \langle \text{set\_stmt} \rangle$   
 $\quad \mid \langle \text{var\_declaration} \rangle$   
 $\quad \mid \langle \text{assignment} \rangle \text{' ;'}$   
 $\langle \text{assignment} \rangle ::= \langle \text{id} \rangle \text{' = ' } \langle \text{exp} \rangle$   
 $\langle \text{brackets\_stmt} \rangle ::= \text{' {' } \langle \text{stmts} \rangle \text{' } \text{' } \text{' '}$   
 $\langle \text{io\_stmt} \rangle ::= \text{read ' (' } \langle \text{id} \rangle \text{' )' ' ;'}$   
 $\quad \mid \text{write ' (' } \langle \text{string} \rangle \mid \langle \text{exp} \rangle \text{' )' ' ;'}$   
 $\quad \mid \text{writeln ' (' } \langle \text{string} \rangle \mid \langle \text{exp} \rangle \text{' )' ' ;'}$   
 $\langle \text{for\_stmt} \rangle ::= \text{for ' (' } \langle \text{assignment} \rangle \text{' ;' } \langle \text{exp} \rangle \text{' ;' } \langle \text{assignment} \rangle \text{' )' } \langle \text{stmt} \rangle$   
 $\langle \text{if\_else\_stmt} \rangle ::= \text{if ' (' } \langle \text{exp} \rangle \text{' )' } \langle \text{stmt} \rangle$   
 $\quad \mid \text{if ' (' } \langle \text{exp} \rangle \text{' )' } \langle \text{brackets\_stmt} \rangle \text{ else } \langle \text{stmt} \rangle$   
 $\langle \text{return\_stmt} \rangle ::= \text{return ' ;' } \mid \text{return } \langle \text{exp} \rangle \text{' ;'}$   
 $\langle \text{set\_stmt} \rangle ::= \text{forall ' (' } \langle \text{id} \rangle \text{ in } \langle \text{set\_exp} \rangle \text{' )' } \langle \text{stmt} \rangle$   
 $\langle \text{exp\_stmt} \rangle ::= \langle \text{exp} \rangle \text{' ;' } \mid \text{' ;'}$   
 $\langle \text{exp} \rangle ::= \langle \text{or\_exp} \rangle \mid \langle \text{set\_exp} \rangle$   
 $\langle \text{set\_exp} \rangle ::= \text{add ' (' } \langle \text{set\_in\_exp} \rangle \text{' )'}$   
 $\quad \mid \text{remove ' (' } \langle \text{set\_in\_exp} \rangle \text{' )'}$   
 $\quad \mid \text{exists ' (' } \langle \text{set\_in\_exp} \rangle \text{' )'}$   
 $\langle \text{set\_in\_exp} \rangle ::= \langle \text{or\_exp} \rangle \text{ in } \langle \text{set\_exp} \rangle$   
 $\langle \text{or\_exp} \rangle ::= \langle \text{or\_exp} \rangle \text{' || ' } \langle \text{and\_exp} \rangle \mid \langle \text{and\_exp} \rangle$   
 $\langle \text{and\_exp} \rangle ::= \langle \text{and\_exp} \rangle \text{' \&\& ' } \langle \text{relational\_exp} \rangle \mid \langle \text{relational\_exp} \rangle$

$$\begin{aligned}
\langle relational\_exp \rangle &::= \langle relational\_exp \rangle \langle relational\_op \rangle \langle sum\_exp \rangle \mid \langle sum\_exp \rangle \\
\langle relational\_op \rangle &::= '<' \mid '>' \mid '>=' \mid '<=' \mid '==' \mid '!=' \\
\langle sum\_exp \rangle &::= \langle sum\_exp \rangle '+' \langle mul\_exp \rangle \\
&\mid \langle sum\_exp \rangle '-' \langle mul\_exp \rangle \\
&\mid \langle mul\_exp \rangle \\
\langle mul\_exp \rangle &::= \langle mul\_exp \rangle '*' \langle primal\_exp \rangle \\
&\mid \langle mul\_exp \rangle '/' \langle primal\_exp \rangle \\
&\mid \langle unary\_exp \rangle \\
\langle unary\_exp \rangle &::= '!'? \langle primal\_exp \rangle \mid '-' \langle primal\_exp \rangle \\
&\mid '!'? \langle id \rangle '(' \arg\_list ')', \\
&\mid '!'? \langle id \rangle '(' ')', \\
&\mid '!'? \text{is\_set} '(' \langle id \rangle \mid \langle set\_exp \rangle ')', \\
\langle primal\_exp \rangle &::= \langle id \rangle \mid \langle const \rangle \mid '(' \langle exp \rangle ')', \\
\langle arg\_list \rangle &::= \langle exp \rangle, \langle arg\_list \rangle \mid \langle exp \rangle \\
\langle type \rangle &::= \langle basic\_type \rangle \mid \langle elem\_type \rangle \mid \langle set\_type \rangle \\
\langle const \rangle &::= \langle int\_const \rangle \mid \langle float\_const \rangle \mid \langle empty\_const \rangle \\
\langle int\_const \rangle &::= \langle digit \rangle + \\
\langle float\_const \rangle &::= \langle digit \rangle + '.' \langle digit \rangle^* \\
\langle empty\_const \rangle &::= 'EMPTY' \\
\langle elem\_type \rangle &::= \text{elem} \\
\langle set\_type \rangle &::= \text{set} \\
\langle int\_type \rangle &::= \text{int} \\
\langle float\_type \rangle &::= \text{float} \\
\langle string \rangle &::= .* \mid '.*' \\
\langle id \rangle &::= [a-zA-Z\_][\_a-z0-9A-Z]^* \\
\langle number \rangle &::= [0-9]
\end{aligned}$$