Analisador 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 `lexical.l` 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 `lexical.l` também foram definidas funções para leitura de arquivo, assim como três variáveis `int line`, `int column` e `int errors`, que representam, respectivamente, a linha em qual está acontecendo a ação, a coluna e o número de erros encontrados até o momento.

O analisador léxico também é responsável pela atribuição dos tokens que serão utilizados durante a análise sintática. Os tokens são declarados como uma `struct Token` que possui os parâmetros `char* body` que consiste no token propriamente dito, `int line` e `int column` 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 `syntax.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 uma DFS 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.

Cada 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.

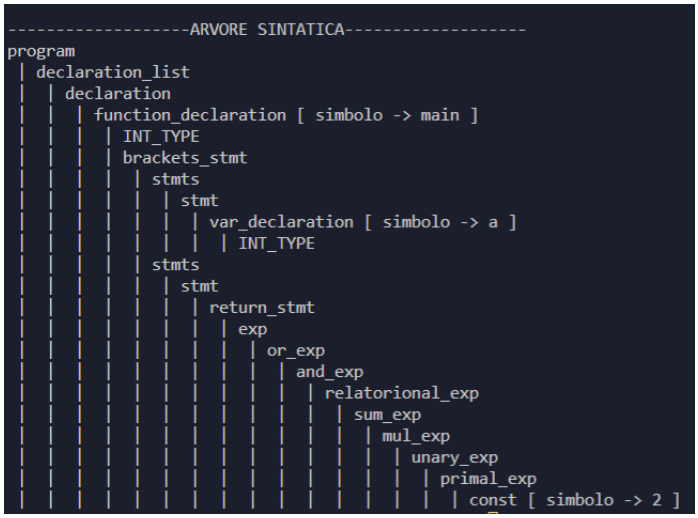

Figura 1. Árvore Sintática para uma entrada simples

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

Acima é possível visualizar a tabela de símbolos para o arquivo `success1.c`.

5 Descrição dos arquivos testes

Os testes se encontram na pasta `tests/`, os arquivos `success1.c` e `success2.c`, são testes que contém código correto, já os arquivos `error1.c` e `error2.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 - `error1.c` - possui tanto erros léxicos, quando erros sintáticos, já o segundo arquivo - `error2.c` - possui apenas erros sintáticos. Quando um arquivo possui erros não são apresentadas a tabela de símbolos nem a árvore sintática. 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 syntax.y
$ flex lexical.l
$ gcc syntax.tab.c lex.yy.c tabela.c arvore.c
$ ./a.out tests/<nome-arquivo>.c
```

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

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

⟨*program*⟩ ::= ⟨*declaration_list*⟩

⟨*declaration_list*⟩ ::= ⟨*declaration*⟩ ⟨*declaration_list*⟩ | ⟨*declaration*⟩

⟨*declaration*⟩ ::= ⟨*function_declaration*⟩ | ⟨*var_declaration*⟩

⟨*var_declaration*⟩ ::= ⟨*type*⟩ ⟨*id*⟩ ';'

⟨*function_declaration*⟩ ::= ⟨*type*⟩ ⟨*id*⟩ '(' ⟨*params_list*⟩ ')' ⟨*brackets_stmt*⟩
| ⟨*type*⟩ ⟨*id*⟩ '(' ')' ⟨*brackets_stmt*⟩

⟨*params_list*⟩ ::= ⟨*type*⟩ ⟨*id*⟩ ',' ⟨*param_list*⟩ | ⟨*type*⟩ ⟨*id*⟩

⟨*stmts*⟩ ::= ⟨*stmt*⟩ ⟨*stmts*⟩ | ⟨*stmt*⟩

⟨*stmt*⟩ ::= ⟨*for_stmt*⟩ | ⟨*if_else_stmt*⟩ | ⟨*return_stmt*⟩ | ⟨*io_stmt*⟩
| ⟨*brackets_stmt*⟩
| ⟨*exp_stmt*⟩
| ⟨*set_stmt*⟩
| ⟨*var_declaration*⟩
| ⟨*assignment*⟩ ';'

⟨*assignment*⟩ ::= ⟨*id*⟩ '=' ⟨*exp*⟩

⟨*brackets_stmt*⟩ ::= '{' ⟨*stmts*⟩ '}'

⟨*io_stmt*⟩ ::= read '(' ⟨*id*⟩ ')' ';'
| write'(' ⟨*string*⟩ | ⟨*exp*⟩ ')' ';'
| writeln'(' ⟨*string*⟩ | ⟨*exp*⟩')' ';'

⟨*for_stmt*⟩ ::= for '(' ⟨*assignment*⟩ ';' ⟨*exp*⟩ ';' ⟨*assignment*⟩ ')' ⟨*stmt*⟩

⟨*if_else_stmt*⟩ ::= if '(' ⟨*exp*⟩ ')' ⟨*stmt*⟩
| if '(' ⟨*exp*⟩ ')' ⟨*brackets_stmt*⟩ else ⟨*stmt*⟩

⟨*return_stmt*⟩ ::= return ';' | return ⟨*exp*⟩ ';'

⟨*set_stmt*⟩ ::= forall'(' ⟨*id*⟩ in ⟨*set_exp*⟩ ')' ⟨*stmt*⟩

⟨*exp_stmt*⟩ ::= ⟨*exp*⟩ ';' | ';'

⟨*exp*⟩ ::= ⟨*or_exp*⟩ | ⟨*set_exp*⟩

⟨*set_exp*⟩ ::= add'(' ⟨*set_in_exp*⟩ ')'
| remove'(' ⟨*set_in_exp*⟩ ')'
| exists'(' ⟨*set_in_exp*⟩ ')'
| '!'? is_set '(' ⟨*id*⟩ | ⟨*set_exp*⟩ ')'

⟨*set_in_exp*⟩ ::= ⟨*or_exp*⟩ in ⟨*set_exp*⟩

⟨*or_exp*⟩ ::= ⟨*or_exp*⟩ '||' ⟨*and_exp*⟩ | ⟨*and_exp*⟩

⟨*and_exp*⟩ ::= ⟨*and_exp*⟩ '`&&`' ⟨*relational_exp*⟩ | ⟨*relational_exp*⟩

⟨*relational_exp*⟩ ::= ⟨*relational_exp*⟩ ⟨*relational_op*⟩ ⟨*sum_exp*⟩ | ⟨*sum_exp*⟩

⟨*relational_op*⟩ ::= '`<`' | '`>`' | '`>=`' | '`<=`' | '`==`' | '`!=`'

⟨*sum_exp*⟩ ::= ⟨*sum_exp*⟩ '`+`' ⟨*mul_exp*⟩
| ⟨*sum_exp*⟩ '`-`' ⟨*mul_exp*⟩
| ⟨*mul_exp*⟩

⟨*mul_exp*⟩ ::= ⟨*mul_exp*⟩ '`*`' ⟨*primal_exp*⟩
| ⟨*mul_exp*⟩ '`/`' ⟨*primal_exp*⟩
| ⟨*unary_exp*⟩

⟨*unary_exp*⟩ ::= ⟨*primal_exp*⟩ | '`!`' ⟨*primal_exp*⟩ | '`-`' ⟨*primal_exp*⟩
| ⟨*id*⟩ '`(`' arg_list '`)`'
| ⟨*id*⟩ '`(`' '`)`'

⟨*primal_exp*⟩ ::= ⟨*id*⟩ | ⟨*const*⟩ | '`(`' ⟨*exp*⟩ '`)`'

⟨*arg_list*⟩ ::= ⟨*exp*⟩, ⟨*arg_list*⟩ | ⟨*exp*⟩

⟨*type*⟩ ::= ⟨*basic_type*⟩ | ⟨*elem_type*⟩ | ⟨*set_type*⟩

⟨*const*⟩ ::= ⟨*int_const*⟩ | ⟨*float_const*⟩ | ⟨*empty_const*⟩

⟨*int_const*⟩ ::= ⟨*digit*⟩+

⟨*float_const*⟩ ::= ⟨*digit*⟩+ '`.`' ⟨*digit*⟩*

⟨*empty_const*⟩ ::= '`EMPTY`'

⟨*elem_type*⟩ ::= elem

⟨*set_type*⟩ ::= set

⟨*int_type*⟩ ::= int

⟨*float_type*⟩ ::= float

⟨*string*⟩ ::= `.*` | '.*'

⟨*id*⟩ ::= [a-zA-Z_][_a-z0-9A-Z]*

⟨*number*⟩ ::= [0-9]