

Analizador Léxico, Sintático e Semântico

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 consistem na implementação do analisador léxico, sintático e semântico 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, variável ou um parâmetro, e o escopo.

```

-----ARVORE SINTATICA-----
program
| function_declaration--- [ identificador -> main ]
|   INT_TYPE
|   stmts
|   | var_declaration--- [ identificador -> x ]
|   |   INT_TYPE
|   | stmts
|   |   assignment--- [ identificador -> x ]--- -> INT
|   |   | sum_exp--- [ operador -> + ]--- -> INT [ CAST -> float2int ]
|   |   |   CONST--- [ constante -> 1.2 ]--- -> FLOAT
|   |   |   | primal_exp--- [ identificador -> x ]--- -> FLOAT [ CAST -> int2float ]
|   |   return_stmt
|   |   | CONST--- [ constante -> 1 ]--- -> INT

```

Figura 1. Árvore Sintática para uma entrada simples

=====TABELA DE SIMBOLOS=====				
= IDENTIFICADOR	LINHA:COLUNA	TIPO	PARAM/VAR/FUNC	ESCOPO =
=====	=====	=====	=====	=====
f	1:5	INT	FUNC	0
x	1:18	INT	PARAM	1
a	1:11	INT	PARAM	1
main	5:5	INT	FUNC	0
x	6:9	INT	VAR	2
s	7:9	SET	VAR	2
a	9:11	FLOAT	VAR	2
=====	=====	=====	=====	=====

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

5 Descrição da análise semântica

Como já se sabe, a análise semântica é responsável por verificar se o programa "faz sentido" ou não, ou seja, verificação de tipos de variáveis, realização de conversões de tipo, verificação de quantidade de parâmetros, se variáveis foram previamente declaradas, entre outras.

Dito isso, como o Bison [DS21] já faz uma passagem por todo o programa na análise sintática, ou seja, percorre uma única vez a árvore abstrata, utilizamos desta mesma passagem para fazer a maior parte das verificações necessárias da análise semântica. Abaixo seguem subseções sobre as principais regras utilizadas para esta etapa.

5.1 Regras de escopo

Para a implementação dos escopos, foi utilizada a estrutura de dados de pilha, como em C não existe uma estrutura pronta para isso, foi criado uma abstração

desta estrutura, utilizando um array junto com um índice, onde o índice aponta sempre para o último elemento do array.

As funcionalidades do escopo são como as esperadas em um programa em C, na Seção 4.2 é possível visualizar a tabela de símbolos para um dos arquivos testes, e nela se encontra o escopo de cada uma das variáveis.

As verificações feitas durante a análise semântica consistem em, verificar se uma variável ou função foi previamente declarada, se não existem múltiplas declarações da mesma variável no mesmo escopo ou múltiplas declarações da mesma função.

5.2 Regras de passagem de parâmetros

Quando uma função é declarada, são associados seus parâmetros a ela, ou seja, cada símbolo da tabela que for uma função, acompanha uma lista de parâmetros. Além disso, os parâmetros são apresentados na tabela de símbolos e propriamente identificados como é possível ver na Seção 4.2.

Ao ser chamada, uma função passa por duas verificações, a primeira consiste em verificar se a quantidade de parâmetros é a mesma da quantidade de parâmetros que foram declarados na criação da função, caso isso não seja correspondido, o programa gera um erro, apresentando quantos parâmetros foram passados e quantos são esperados para aquela função.

A segunda verificação, é a análise dos tipos, ela será mais abordada na Seção 5.3, mas rapidamente, o que ela faz é, verificar se os tipos dos parâmetros correspondem aos tipos declarados na função, se os tipos não forem iguais, tenta-se fazer a conversão de tipos, caso isso não seja possível, por incompatibilidade dos tipos, um erro é lançado, apresentando o tipo passado e o tipo esperado.

5.3 Decisões sobre conversões de tipos

Como apresentado anteriormente, a motivação para esta linguagem é facilitar a utilização de conjuntos. Dito isso, os tipos presentes na linguagem são `int`, `float`, `set` e `elem`, sendo os dois últimos mais específicos para a utilização de conjuntos.

Para realizar as coersões necessárias, foram adicionadas informações de tipos para cada nó da árvore, como é possível visualizar na Figura 1. Quando há uma coersão de tipo, ela é sinalizada como `CAST` e em seguida é especificado qual coersão foi realizada, por exemplo, `int2float` significa que o nó era do tipo `int` e que foi convertido para o tipo `float`.

Alguns tipos e funções tem suas particularidades, por exemplo, não é possível converter um `int` ou um `float` para o tipo `set`, visto que são tipos muito distintos. Abaixo seguem algumas das regras específicas para cada tipo:

- `int` pode ser convertido para `float`, e vice-versa;
- O tipo `elem` pode ser convertido para qualquer um dos outros tipos, e vice-versa;

- A constante `EMPTY` é tratada como vazio (`NULL`) e só pode ser utilizada em tipos `elem` ou `set`;
- Funções como `is_set` ou para verificação de uma variável dentro de um `set` (i.e: `1 in s`) retornam o tipo `int`.
- Funções como `add` e `remove` retornam o tipo `set` e na parte direita dessas expressões sempre é esperado o tipo `set`;
- Não é possível realizar operações aritméticas com o tipo `set`;

6 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. SEMANTIC ERROR 3:9 - Incompatible types can't be
casted -> INT and SET
SEMANTIC ERROR 3:5 - Incompatible types can't be
casted -> SET and INT
SEMANTIC ERROR 4:7 - Redeclaration of variable 'b'
SEMANTIC ERROR 10:10 - `is_set` only supports the type SET
SEMANTIC ERROR 13:3 - Fewer number of arguments in
function 'a' - expected: 2 , received: 0
SEMANTIC ERROR 14:3 - Greater number of arguments in
function 'a' - expected: 2 , received: 3
SEMANTIC ERROR - The program doesn't have a 'main' function
2. LEXICAL ERROR 2:8 - Unidentified character: #
SYNTAX ERROR 3:8 - syntax error, unexpected ';',
expecting '='
SYNTAX ERROR 3:14 - syntax error, unexpected ')'
SYNTAX ERROR 9:3 - syntax error, unexpected INT_TYPE
SEMANTIC ERROR 11:5 - Incompatible types can't be
casted -> INT and SET

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 apenas erros semânticos, já o segundo arquivo - `erro2.c` - possui tanto erros léxicos, quanto erros sintáticos e semânticos.

Quando um arquivo possui apenas erros semânticos é apresentada a árvore sintática, porém caso existam erros sintáticos ou léxicos ela não é impressa. A tabela de símbolos gerada (que pode não estar completa dependendo do tipo dos erros) é sempre apresentada, independente do número de erros. Além disso, cada erro é apresentado seguido de uma descrição do mesmo.

7 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

$$\begin{aligned}
\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{'('} \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{'}' } \\
\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_aux_exp} \rangle &::= \langle \text{id} \rangle \text{'in' } \langle \text{set_exp} \rangle \mid \langle \text{id} \rangle \text{'in' } \langle \text{or_exp} \rangle \\
\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
\end{aligned}$$

$$\begin{aligned}
\langle and_exp \rangle &::= \langle and_exp \rangle \text{ '&&' } \langle relational_exp \rangle \mid \langle relational_exp \rangle \\
\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 &::= \text{'<'} \mid \text{'>'} \mid \text{'>='} \mid \text{'<='} \mid \text{'=='} \mid \text{'!='} \\
\langle sum_exp \rangle &::= \langle sum_exp \rangle \text{'+'} \langle mul_exp \rangle \\
&\mid \langle sum_exp \rangle \text{'-'} \langle mul_exp \rangle \\
&\mid \langle mul_exp \rangle \\
\langle mul_exp \rangle &::= \langle mul_exp \rangle \text{'*'} \langle primal_exp \rangle \\
&\mid \langle mul_exp \rangle \text{'/'} \langle primal_exp \rangle \\
&\mid \langle unary_exp \rangle \\
\langle unary_exp \rangle &::= \text{'!'}? \langle primal_exp \rangle \mid \text{'-'} \langle primal_exp \rangle \\
&\mid \text{'!'}? \langle id \rangle \text{'('} \text{arg_list} \text{')'} \\
&\mid \text{'!'}? \langle id \rangle \text{'('} \text{' ')} \\
&\mid \text{'!'}? \text{is_set} \text{'('} \langle id \rangle \mid \langle set_exp \rangle \text{')'} \\
\langle primal_exp \rangle &::= \langle id \rangle \mid \langle const \rangle \mid \text{'('} \langle exp \rangle \text{')'} \\
\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 + \text{'.'} \langle digit \rangle^* \\
\langle empty_const \rangle &::= \text{'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 &::= \text{'.'} * \mid \text{'.'} * \\
\langle id \rangle &::= [\text{a-zA-Z_}][_ \text{a-z0-9A-Z}]^* \\
\langle number \rangle &::= [0-9]
\end{aligned}$$