Analisador Sintático

Linguagem Wladus - Trabalho de Tradutores

Wladimir Gramacho - 15/0048718

Profª. Cláudia Nalon

Setembro de 2019

1 Objetivos

O curso de Tradutores ministrado na Universidade de Brasília tem como objetivo o estudo dos componentes de um tradutor assim como a implementação destes[1]. Sendo assim, o trabalho desta matéria consiste na construção de um tradutor para uma linguagem de programação simplificada e de propósito específico. A linguagem que está sendo criada se chama Wladus e tem como objetivo possibilitar a interpolação de strings e criação de strings como tipo nativo da linguagem. O desenvolvimento do projeto se dá em 6 etapas diferentes, sendo elas: Escolha do Tema, Analisador Léxico, Analisador Sintático, Analisador Semântico, Gerador de Código Intermediário e Apresentação do Trabalho. Este relatório se trata da apresentação da segunda fase: Analisador Sintático.

2 Desenvolvimento

Algumas etapas são necessárias para a construção de um analisador sintático. Após a criação dos *tokens* no analisador sintático, é estabelecida a comunicação entre léxico e sintático por meio desses *tokens*. Com isso, criou-se a tabela de símbolos, onde foram armazenadas informações sobre variáveis e funções. Uma vez feitas essas etapas, foi iniciado o processo de desenvolvimento da principal entrega deste trabalho: a árvore sintática. Foi usado o flex[2] para análise léxica e o bison[3] para análise sintática.

A árvore sintática foi construída em camadas. A cada iteração, parte da árvore era montada, seguindo uma construção *top-down*. Dessa forma, os primeiros nós adicionados são referentes às primeiras regras da gramática. De acordo com a necessidade, foram criados diferentes tipos de nó para a árvore sintática. Alguns exemplos são: nó para a declaração de função, nó para estrutura de iteração e nó para uma instância de um

[1] `https://matriculaweb.unb.br/graduacao/disciplina.aspx?cod=116459`
[2] `https://github.com/westes/flex`
[3] `https://www.gnu.org/software/bison/`

número inteiro. Essa ideia de implementação foi inspirada pelo tutorial do blog do Efstathios Chatzikyriakidis[4].

Foram feitas também funções auxiliares para a exibição da árvore sintática e da tabela de símbolos e liberação da memória alocada pela árvore sintática e tabela de símbolos. A exibição é feita na saída padrão e pode ser habilitada passando um argumento ao executar o programa. Essas funções ainda serão modularizadas para outro arquivo auxiliar para a melhor legibilidade do código.

2.1 Tabela de Símbolos

A tabela de símbolos foi construída no formato de uma *hash table*. Para isso, foi utilizada a biblioteca uthash[5] para o auxílio na implementação. São armazenadas as seguintes informações na tabela de símbolos:

- Nome do identificador. Esse campo é utilizado como chave da *hash table*, de modo que seja possível procurar diretamente pelo nome do identificador na tabela de símbolos, obtendo uma complexidade de tempo constante para busca e inserção na tabela.
- Tipo do identificador (int, float, void ou string).
- Tipo do object (variável ou função).
- Ponteiro para nó na árvore sintática. Esse campo é usado para que seja possível acessar o corpo de uma função declarada.
- *Handle* necessário para a criação da *hash table* segundo a implementação do uthash.

2.2 Árvore Sintática

Como dito anteriormente, a árvore sintática foi criada em camadas, com diferentes tipos de nós sintáticos. Dessa forma, foi possível armazenar diferentes dados em cada nó, inerentes à estrutura da linguagem.

2.3 Dificuldades

Além das dificuldades inerentes ao desenvolvimento de uma linguagem, um obstáculo foi entender como seria feita a transferência da sequência de *tokens* para o analisador sintático. Após esse entendimento, foi necessário muito tempo para entender o processo de construção da árvore sintática. O autor não mensurou corretamente a quantidade de tempo necessária para esta etapa e algumas funcionalidades não puderam ser implementadas a tempo.

[4] `https://efxa.org/2014/05/25/how-to-create-an-abstract-syntax-tree-while-parsing-an-input-stream/?unapproved=60685&moderation-hash=6be251ef24ef24f13ecfd20facfcd2fc#comment-60685`

[5] `https://troydhanson.github.io/uthash/`

2.4 Gramática

A gramática implementada no analisador sintático sofreu algumas mudanças em relação à apresentada no analisador léxico. Essa gramática é inspirada na gramática do padrão C ANSI apresentada por Jutta Degener em 1995, disponível no site da Linkoping University[6]. As mudanças feitas são:

- Adição do tipo de dados para números reais. Foi necessário adicionar um novo tipo ($float$) e identificar esse número nas expressões regulares da linguagem.
- Adição das funções nativas *read* e *write* para leitura e escrita, respectivamente.
- Remoção do suporte a vetores na linguagem. Isso se deu por limitação de tempo e o autor acredita que para o propósito do trabalho e da linguagem criada, não é necessária a implementação de vetores.
- Correção na chamada de função, onde antes era possível passar uma atribuição de uma variável como argumento de uma função. Na regra *arg-list* foi trocado *expression* por *simple-expression*.
- A variável *type* virou um terminal (**TYPE**) que armazena o tipo de um variável ou função.
- Remoção da necessidade de uma regra intermediária para implementar precedência. A precedência é feita através de comandos no arquivo do analisador sintático. Com isso, foi removida a regra $factor$.
- Remoção do comentário da gramática. Agora, *whitespace* e *comment* são somente expressões regulares no léxico que ignoram a entrada.
- Renomear *declaration-list* para *declarations*.
- Renomear fun-*declaration* para $func$-*declaration*.

Portanto, a gramática atual tem a forma:

1. *program* → *declarations*
2. *declarations* → *declarations declaration* | *declaration*
3. *declaration* → *var-declaration* | $func$-*declaration*
4. *var-declaration* → **TYPE ID ;**
5. fun-*declaration* → **TYPE ID (** *params* **)** *compound-stmt*
6. *params* → *param-list* | ε

[6] `https://www.lysator.liu.se/c/ANSI-C-grammar-y.html`

7. *param-list* → *param-list* **,** *param* | *param*
8. *param* → **TYPE ID**
9. *compound-stmt* → **{** *local-declarations stmt-list* **}**
10. *local-declarations* → *local-declarations var-declaration* | ε
11. *stmt-list* → *stmt-list stmt* | ε
12. *stmt* → *expression-stmt* | *conditional-stmt* | *iteration-stmt* | *return-stmt*
13. *expression-stmt* → *expression* **;**
14. *conditional-stmt* → **if (** *expression* **)** *compound-stmt* |
    if (*expression* **)** *compound-stmt* **else** *compound-stmt*
15. *iteration-stmt* → **while (** *expression* **)** *compound-stmt*
16. *return-stmt* → **return** *expression* **;** | **return ;**
17. *expression* → *var* **=** *expression* | *simple-expression*
18. *var* → **ID**
19. *simple-expression* → *op-expression relop op-expression* | *op-expression*
20. *relop* → **<=** | **<** | **>** | **>=** | **==** | **!=**
21. *op-expression* → *op-expression op term* | *term*
22. *op* → **+** | **-** | **/** | *****
23. *term* → **(** *expression* **)** | *var* | *call* | **“** *string* **”** | **NUM** | **DEC**
24. *call* → **ID (** *args* **)** | **write (** *simple-expression* **)** | **read (** *var* **)**
25. *string* → *string* **STRING** | *string* **#{** *expression* **}** | ε
26. *args* → *arg-list* | ε
27. *arg-list* → *arg-list* **,** *simple-expression* | *simple-expression*

ID = $letter\ (letter|digit)^*$

NUM = $digit^+$

DEC = $(digit)^+.(digit)^+$

TYPE = $(int|float|void|string)$

STRING = $(\backslash\backslash.|[^{\wedge}\#\{\}\backslash\backslash])^*$

$letter = a \mid ... \mid z \mid A \mid ... \mid Z$

$digit = 0 \mid ... \mid 9$

$whitespace = \backslash n \mid \ \mid \backslash t$

$comment = "//".*$

2.5 Tratamento de Erros

Por limitação de tempo, não foi possível implementar o tratamento de erros nesta etapa. Os arquivos de teste com erros também foram ignorados para a execução dos testes no makefile.

3 Arquivos de teste

Junto com os analisadores léxico e sintático e o arquivo de diretivas para execução (*makefile*), os arquivos compactados enviados na submissão dessa etapa contêm arquivos de teste do analisador. Há três arquivos de teste com entradas válidas (valid0.w, valid1.w e valid2.w). Para execução do makefile, entre na pasta `src` e execute no terminal de comando as seguintes linhas:

```
make all
```

```
make test
```

A primeira linha compila os analisadores, criando um arquivo executável chamado `prog`. A segunda linha executa três vezes o arquivo `prog` recebendo como entrada os três arquivos válidos. A execução do `prog` tem uma *flag* `-tt` que habilita a impressão da tabela de símbolos e da árvore sintática.