Construção de Compilador utilizando Python

Ricardo A. Müller[1], **Tiago A. Debastiani**[1]

[1]Centro de Computação – Universidade Federal da Fronteira Sul (UFFS)
Caixa Postal 15.064 – 91.501-970 – Chapecó – SC – Brazil

Resumo. *Este trabalho descreve a construção de um compilador, seguindo todas as suas etapas de compilação utilizando a linguagem Python, como trabalho da disciplina de Compiladores da Universidade Federal da Fronteira Sul.*

1. Introdução

Segundo [Aho et al. 1995], um compilador é basicamente um programa que lê um programa em uma determinada linguagem (chamada de *linguagem fonte*) e o traduz em um programa equivalente na *linguagem alvo*. Neste processo o compilador deverá, caso exista, retornar ao usuário os erros existentes no código fonte.

O trabalho realizado visa a implementação das etapas de compilação, sendo elas a léxica, sintática, semântica, gerador de código intermediário e otimização, a fim de construir um programa compilador através de uma Gramática Livre de Contexto (GLC).

Este artigo está organizado da seguinte forma. No Capítulo 2 é apresentado o referencial teórico básico necessário para o entendimentos deste trabalho, explicando o que são as etapas realizadas por um compilador. O Capítulo 3 está dividido em seções que detalham passo-a-passo a execução do programa. Por fim, o Capítulo 4 resume as informações apresentada neste trabalho.

2. Referencial Teórico

A etapa de análise léxica é responsável por identificar os *tokens* da linguagem, a fim de encontrar elementos não reconhecidos na linguagem. Na análise sintática, verifica-se a função de cada elemento no contexto da sentença, para avaliar se as sentenças formadas pela junção dos *tokens* é válida para a linguagem definida. A terceira etapa trata-se da análise semântica, que verifica se o significado dos *tokens* é respeitado, ou seja, nesta etapa são feitas as verificações quanto a tipagem dos dados.

Terminadas as etapas de análise sintática e semântica, o código intermediário é gerado. Podemos, segundo [Aho et al. 1995], pensar no código intermediário como um programa para uma máquina abstrata, que deve possuir duas propriedades importantes, ser fácil de produzir e fácil de traduzir para o programa alvo.

Por último, realiza uma otimização no código intermediário, buscando melhorar o desempenho através da reordenação dos processos à serem executados e a melhor utilização da memória.

Utilizando a linguagem *Python*, este trabalho busca implementar todas as etapas de análise, além da geração de código intermediário e otimização. Seus detalhes estarão descritos no decorrer deste artigo.

3. Desenvolvimento

O projeto inicia definindo os *tokens* através das produções definidas por uma Gramática Regular (GR)[1], gerando um Autômato Finito Determinístico, utilizado para reconhecer os *tokens* do código fonte[2] entradas na análise léxica.

Utilizando-se da notação BNF, a Gramática Livre de Contexto (GLC)[3] possui um símbolo inicial `<S>` e dois *tokens* pré definidos: `var`, que representa a definição de variáveis, ou seja, o conjunto de caracteres subsequente ao *token* será tratado como variável; e `id`, que define constantes numéricas. As etapas seguintes preocupam-se em verificar a estrutura dos dados de entrada, sendo a primeira destas etapas, a análise sintática.

3.1. Análise Sintática

Considerando a GLC acima citada, este compilador gera a tabela SLR, utilizando a ferramenta *Gold Parser*, que recebe a GLC e retorna o mapeamento da gramática. O arquivo XML deste retorno então é lido e forma a tabela SLR. Neste arquivo encontram-se os símbolos terminais e não-terminais, saltos, reduções e transições, essenciais para o processo de mapeamento da linguagem.

Subsequente a isto, será necessário percorrer a tabela de símbolos, gerada pela análise léxica, para que, juntamente com o estado atual da pilha, possamos fazer o mapeamento na tabela SLR. Sendo assim a tabela SLR irá conter informações sobre a operação a ser realizada, seja ela um salto, redução ou transição de estado.

Caso o mapeamento indique um estado não esperado no processo, determina-se um erro sintático. Caso o erro seja constatado, a aplicação retornará um aviso sobre o *token* e a linha onde o erro ocorreu. Caso contrário, a tabela retornará um estado de aceitação, a aplicação concluirá a análise sintática, gerando uma nova tabela de símbolos, com informações sobre o *token* e seguirá para a análise semântica.

3.2. Análise Semântica

Através da Tabela de Símbolos, a Análise Sintática busca por atribuições de variáveis, retornando erro caso haja atribuição de variáveis de tipos distintos, exemplo $a = b$, onde a é do tipo inteiro e b do tipo booleano.

Neste projeto, validações semânticas foram aplicadas apenas à alguns aspectos da linguagem, como as produções referentes a gramática de operadores, declaração de variável e atribuições de modo geral.

Ao final da análise, caso não ocorra nenhum erro, é gerado um arquivo contendo o código intermediário do código fonte[4].

3.3. Código Intermediário

Derivada do resultado da análise semântica, esta etapa apenas é gerada caso a análise sintática e semântica ocorram sem erros.

[1] Ver Apêndice A.
[2] Ver Apêndice C.
[3] Ver Apêndice B
[4] Ver Apêndice D.

O código intermediário utilizado segue a abordagem ”código de três endereços”, ou seja, há uma sequência de instruções, onde cada instrução possui no máximo três operandos. Para operações matemáticas deve-se seguir a precedência dos operandos no momento de distribuir as operações, como por exemplo, a multiplicação deve ser realizada antes da soma.

Diversas variáveis temporárias devem ser criadas para realizar todas as operações do código fonte. Quando cada operação for finalizada, gera um código, que é adicionada a uma lista. Sendo assim, o código intermediário é composto por cada item desta lista.

3.4. Otimização

Utilizando o código intermediário, gerado na etapa anterior, este projeto utiliza a otimização apenas nas operações aritméticas, ou seja, declarações e atribuições não serão otimizadas.

Para a otimização, um grafo acíclico é gerado, onde os nodos folhas do grafo são formados pelo operandos e os nodos internos são os temporários, que corresponde as operações. Agora, buscando otimizar a ordem das operações, utiliza-se neste projeto a busca em profundidade, com orientação à esquerda. Sendo assim, o primeiro nodo a ser visitado será o primeiro nodo no topo mais à esquerda que existir.

Cria-se então uma lista de nodos visitados, onde um nodo só é adicionado se respeitas as restrições do problema, caso contrário a busca continuará sem o incluir na lista. Estando todos os nodos na lista de visitados, deve-se percorrer a lista do maior índice para o menor (ordem inversa à adição), gerando assim a nova ordem de processos e, consequentemente, o código otimizado[5].

4. Conclusão

Este trabalho mostra a implementação das etapas de um compilador, utilizando a linguagem *Python*. O projeto cria um AFD a partir de uma GR, após isso, realiza a leitura sequencial do arquivo fonte, além do auxílio da ferramenta *Gold Parser* (que gera as tabelas *parsing*), fazendo as validações léxicas, sintáticas e semânticas definidas, além de geração de código intermediário e otimização. O trabalho conseguiu alcançar o objetivo proposto, segundo as limitações apresentadas no Seção 3.2, mas o código intermediário e a otimização foram realizadas para as mesmas.

Um trabalho futuro à esta implementação, seriam a complementação das validações semânticas, além da geração do código intermediário e otimização para as mesmas.

Referências

[Aho et al. 1995] Aho, A. V., Sethi, R., and Ullman, J. D. (1995). *Compiladores. Princípios, Técnicas e Ferramentas*. LTC - Livros Técnicos e Científicos Editora S.A.

[5] Ver Apêndice E.

A. Gramática Regular

```
if
else
while
TRUE
FALSE
MAX
+
-
*
/
%
)
(
integer
real
boolean
return
{
}
==
!=
>
<
<=
>=
;
&&
||
=

<S> ::= a<A> | b<A> | c<A> | d<A> | e<A> | f<A> | g<A> | h<A> |
    ↪  i<A> | j<A> | k<A> | l<A> | m<A> | n<A> | o<A> | p<A> | q<A> |
    ↪  r<A> | s<A>| t<A> | u<A> | v<A> | w<A> | x<A> | y<A> | z<A> |
<A> ::= a<A> | b<A> | c<A> | d<A> | e<A> | f<A> | g<A> | h<A> |
    ↪  i<A> | j<A> | k<A> | l<A> | m<A> | n<A> | o<A> | p<A> | q<A> |
    ↪  r<A> | s<A>| t<A> | u<A> | v<A> | w<A> | x<A> | y<A> | z<A> | ϵ
<S> :: = 0<B>| 1<B>| 2<B>| 3<B>| 4<B>| 5<B>| 6<B>| 7<B>| 8<B>|
    ↪  9<B>|
<B> :: = 0<B>| 1<B>| 2<B>| 3<B>| 4<B>| 5<B>| 6<B>| 7<B>| 8<B>|
    ↪  9<B>| ϵ
```

B. Gramática Livre de Contexto

```
"Start Symbol" = <S>

var = {Letter}{alphaNumeric}*
id = {Digit}+

<S>::= <IF> <S> | <ELSE> <S> | <WHILE> <S> |  <OPER> <S> | <DEC>
    ↪  <S> | 'return'';' | <CHAIN_IF>
<if>::= 'if''('<COND>')''{'<S>
<CHAIN_IF>::= '}'
<ELSE>::= 'if''('<COND>')''{'<S>'else''{'<S>
<WHILE>::= 'while''('<COND>')''{'<S>
```

```
<COND>::= 'id' | 'id'<OP_LOGIC><COND> | <CONSTANT> | <CONSTANT>
    ↪  <OP_LOGIC> <COND> | 'var' | 'var' <OP_LOGIC> <COND>
<OP_LOGIC>::= '==' | '!=' | '<' | '>' | '<=' | '>=' | '&&' | '||'
<OPER> ::= 'var''=' <E>';' | 'var' '=' 'var'';' |'var' '='
    ↪  <CONSTANT>';'
<E> ::= <E>'+'<T> | <E>'-'<T> | <T>
<T> ::= <T>'*'<F> | <T>'/'<F> | <F>
<F> ::= '('<E>')'| 'var' | <CONSTANT> | 'id'
<DEC>::= <TYPE> 'var'';'
<TYPE>::= 'integer' | 'real' | 'boolean'
<CONSTANT>::= 'TRUE'| 'FALSE'| 'MAX'
```

C. Código Fonte

```
integer a;
integer b;
a= b * (a-999) / a ;
a = b;
if(a < b){
}else{
        a = 20;
}
return;
```

D. Código Intermediário

```
integer a
integer b
temp0 = a - 999
temp1 = b * temp0
temp2 = temp1 / a
a = b
a = 20
```

E. Código Otimizado

```
integer a
integer b
a = b
a = 20
temp0 = a - 999
temp1 = b * temp0
temp2 = temp1 / a
```