Reconhecimento Sintático em Compiladores

Gabriel H. Rudey[1], **Jefferson A. Coppini**[1], **Jonathan T. Rauber**[1], **Nicholas S. Brutti**[1], **Ricardo A. Müller**[1]

[1]Ciência da Computação – Universidade Federal da Fronteira Sul (UFFS)
Caixa Postal 181 - CEP 89802-112 – Chapecó – SC – Brazil

Resumo. *Este trabalho descreve o processo de implementação de um analisador sintático com geração de código intermediário e otimização na linguagem Python como trabalho da disciplina de Compiladores da Universidade Federal da Fronteira Sul, sob orientação do professor Bráulio Adriano de Mello.*

1. Introdução

Toda linguagem de programação possui uma estrutura sintática particular bem definida, onde um erro de digitação ou o não seguimento das regras da gramática implica na falha de compilação. Nesse contexto, o reconhecimento sintático tem como propósito verificar se as cadeias de *tokens* produzidas pelo analisador léxico podem ser geradas pela linguagem-fonte [AHO et al. 2007].

Este trabalho visa a implementação de um analisador sintático para uma linguagem hipotética – definida através de uma Gramática Livre de Contexto (GLC) durante o desenvolvimento – com análise léxica, geração de código intermediário e otimização de algumas das estruturas contidas na linguagem.

2. Referencial Teórico

Entende-se "Sintaxe" como a ciência da língua que estuda o processo de construção das sentenças. Nessa perspectiva, realizar a análise sintática significa identificar a função que cada elemento exerce no contexto da sentença, bem como a relação que ele estabelece com os demais constituintes, de modo a se formar um todo organizado e harmônico [FONSECA 2012].

A etapa seguinte é denominada análise semântica, cujo objetivo é verificar se o significado do *token* é respeitado, porque o código pode estar sintaticamente correto mas não é garantido que a semântica também esteja.

Após concluir a análise sintática e semântica, o compilador gera uma representação intermediária explícita do código-fonte, com base na estrutura hierárquica da análise sintática. Pode-se pensar nesse trecho como um programa para uma máquina abstrata. A representação intermediária utilizada foi o "código de três endereços", que consiste em uma sequência de instruções, cada uma delas possuindo no máximo três operandos [AHO et al. 2007].

A etapa de otimização busca melhorar o código intermediário obtido anteriormente, com objetivo de garantir um melhor desempenho. O algoritmo de otimização usado caracteriza-se por detectar segmentos sequenciais do programa, chamados blocos básicos [PRICE and TOSCANI 2008].

Esse trabalho apresenta a implementação de um analisador sintático, com análise semântica, geração de código intermediário e otimização. O código foi escrito em linguagem *Python* e seus detalhes estão descritos nas próximas seções.

2.1. Tabela de símbolos

Importante ressaltar que toda a informação sobre os *tokens* é registrada em um estrutura de dados, conhecida como tabela de símbolos. Essa estrutura é extremamente importante para o funcionamento do compilador, pois, interage diretamente com todas as etapas do processo de compilação, seja para consulta ou inserção. Uma entrada na tabela de símbolos é da forma: $<$*nome_token*, *valor_atributo*$>$. Onde *nome_token* é um símbolo abstrato que representa o identificador, e *valor_atributo* aponta para a entrada correspondente na tabela de símbolos.

2.2. Gramática livre de contexto

Como citado na seção 1, a linguagem hipotética foi definida com uma gramática livre de contexto que possui este nome, segundo [Menezes 1998, p. 86], pois permite a derivação de qualquer produção sem depender de qualquer análise dos símbolos que antecedem ou sucedem a variável que dá nome a regra, sendo assim versátil o suficiente para definição de estruturas sintáticas e aplicadas em projetos de linguagens de programação [PRICE and TOSCANI 2008].

2.3. Analisadores LR

Segundo [PRICE and TOSCANI 2008] os analisadores LR *(Left to right with Rightmost derivation)* são analisadores redutores eficientes que lêem a sentença em análise da esquerda para a direita e produzem uma derivação mais à direita ao reverso. O analisador utilizado é o SLR *(Simple LR)*, cuja principal característica é a facilidade de implementação, porém é restrito a poucas gramáticas.

3. Desenvolvimento

Após a definição dos *tokens* da linguagem através dos carregamento de um arquivo externo e a geração de um autômato finito, sendo este responsável pelo reconhecimento dos dados de entrada durante a análise léxica, o foco a parir de agora não está em dizer se determinada entrada de um arquivo fonte está correta ou não, e sim verificar a estruturalmente os dados de entrada. A primeira etapa responsável por essa ideia é a análise sintática.

3.1. Gramática

A gramática utilizada para desenvolvimento da linguagem, listada na seção de anexo (5) como gramática 1, utiliza a notação BNF e, portanto, possui o símbolo inicial como o primeiro símbolo não-terminal definido, neste caso $\langle S \rangle$, e dois *tokens* pré definidos que são originários da análise léxica: “var” e “id”, onde o primeiro representa uma variável, ou seja, um conjunto de caracteres iniciado por uma letra concatenado com o fechamento transitivo do conjunto alfa-numérico e o segundo – “id” – são constantes numéricas.

3.2. Análise sintática

Dada a gramática livre de contexto citada em seções anteriores, o passo inicial para a criação da etapa de análise sintática de um compilador é a geração da tabela SLR. Para geração desta tabela utiliza-se a ferramenta *Gold Parser*, responsável por receber uma gramática livre de contexto e retornar um arquivo contento todo mapeamento da gramática.

A tabela SLR da aplicação foi formada a partir da leitura de um arquivo *XML* gerado pelo *Gold Parser*. Nesse arquivo existem os símbolos(terminais e não terminais), saltos, reduções e transições necessárias para o processo de mapeamento da linguagem.

Com a tabela construída, o próximo passo consiste em percorrer a tabela de símbolos gerada durante a analise léxica, pois nela estão os *tokens* reconhecidos e que podem ter algum mapeamento na tabela SLR.

As ações de reconhecimento durante o processo de análise é feito de forma a pegar o *token* da tabela de símbolos e o estado atual da pilha de reconhecimento, fazendo assim o mapeamento na tabela SLR. Com isso a tabela irá informar que tipo de operação será feita, seja ela uma redução, salto ou transição de estados. Vale citar que em casos de redução, são verificadas as ações semânticas como será citado posteriormente.

O mapeamento pode informar um estado não esperado durante o processo, caso isso ocorra sera determinado erro sintático. Se a tabela retornar um erro sintático, a aplicação informa em que *token* da tabela de símbolos esse erro ocorreu, em qual linha do arquivo fonte esse *token* se encontra e interrompe o processo de compilação. Caso a tabela retorne um estado de aceitação a aplicação informa que a análise sintática está concluída, e o processo decorre para análise semântica.

3.3. Ações semânticas

O processo de construção das ações semânticas de uma linguagem talvez seja a etapa mais complexa do desenvolvimento pois infere diretamente em ações posteriores no processo de compilação. Especificamente para a gramática livre de contexto desta aplicação, apenas algumas ações semânticas foram implementadas.

Os tipos de ações semânticas implementadas fazem relação a alguns aspectos importantes da gramática. Uma delas é em relação as produções referentes a gramática de operadores, outro é em relação a declaração de variáveis e por último atribuições em geral. Em todos esses casos, ações semânticas foram criadas, suas derivações geraram código intermediário e por consequência aspectos das operações são adicionadas a tabela de símbolos.

3.4. Análise semântica

Após o processo sintático o compilador deve fazer algumas análises em relação a aspectos que as outras verificações são incapazes de fazer. Na aplicação a análise semântica feita foi em relação a atribuição de variáveis. Nesse contexto ocorrerá erro onde dada uma atribuição $x = y$, o tipo da variável x for diferente da variável y. Sendo o mesmo decorrente de declarações anteriores no próprio código-fonte.

3.5. Geração de código intermediário

Essa etapa é indexada a etapas anteriores, relacionando as ações semânticas e por consequência o processo de análise sintática. Por não ser uma ação isolada no sistema, essa etapa pode não ser executada na aplicação caso ocorra algum erro sintático durante a compilação.

Cada variável temporária, inicialmente é produzida durante a finalização das produções nas ações semânticas. Cada operação criada é armazenada a uma pilha de operações, então na criação do temporário atual são usadas as operações dessa pilha. Quando a operação é finalizada, o código gerado é adicionado a uma lista. Após o término da análise sintática o código intermediário será composto por cada item desta lista de operações.

3.6. Otimização

Com o código intermediário gerado, a etapa final da aplicação consiste na otimização do código. A estratégia usada nesse contexto foi fazer otimização apenas das operações aritméticas, não incluindo assim as operações de declaração de variáveis e atribuições.

A primeira etapa da otimização é a construção do grafo acíclico dirigido, onde nas folhas estão os operandos e nos nodos internos os temporários com as operações. Na aplicação esse processo ocorreu percorrendo o arquivo do código intermediário, e determinando os folhas e nodos internos conforme a estrutura.

O objetivo então é fazer uma busca para declarar a ordem em que as operações devem ser escritas. A técnica utilizada na aplicação foi a busca em profundidade, onde o grafo seria percorrido sempre visitando o nodo mais a esquerda. Se o nodo respeita as restrições do problema então é adicionado a lista de nodos visitados, caso contrário a busca continuará sem incluir o nodo na lista.

Para a geração do código intermediário otimizado, deve-se percorrer a lista de nodos visitados na ordem do maior índice até o menor, onde assim estarão as operações expostas em nova ordem.

4. Conclusão

Este trabalho mostra a implementação de um reconhecedor sintático para uma linguagem de programação hipotética desenvolvido sobre a linguagem *Python*, realizando à partir da leitura sequencial do arquivo fonte, a análise sintática, semântica, geração de código intermediário e otimização com auxílio da ferramenta *Gold Parser* para geração das tabelas de *parsing*. O resultado final, geração do código intermediário otimizado, para as estruturas definidas na seção 3.3 foi alcançado, sendo assim um trabalho futuro a implementação de códigos intermediários para as demais produções e geração de código de máquina.

Referências

AHO, A. V., SETHI, R., and LAM, M. (2007). *Compiladores: Princípios, técnicas e ferramentas*. Longman do Brasil.

FONSECA, D. L. (2012). Análise sintática. `http://www.infoescola.com/portugues/analise-sintatica/`. Acessado em: 14/12/2017.

Menezes, P. B. (1998). *Linguagens formais e autômatos*. Sagra-Dcluzzato.

PRICE, A. M. A. and TOSCANI, S. S. (2008). *Implementação de Linguagens de Programação: Compiladores*. Bookman Companhia, 1st edition.

5. Anexos

⟨*S*⟩ ::= ⟨*IF*⟩ ⟨*S*⟩ | ⟨*ELSE*⟩ ⟨*S*⟩ | ⟨*WHILE*⟩ ⟨*S*⟩
| ⟨*OPER*⟩ ⟨*S*⟩ | ⟨*DEC*⟩ ⟨*S*⟩ | ⟨*CHAIN_IF*⟩
| 'return' ';'

⟨*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'

Gramática 1. Gramática utilizada no programa