COMPILADORES

Universidade Estadual do Piauí - UESPI | Campus Parnaíba

Proposta da Atividade

Desenvolver os analisadores Léxico, Sintático e Semântico. A finalidade é analisar um código fonte informado pelo usuário e retornar a validação das informações passadas ou algum erro, caso seja identificado. Além disso, o desenvolvimento dessa atividade busca fixar o conhecimento sobre o funcionamento de um compilador em partes.

O que é um compilador ?

É um programa que traduz um programa descrito em uma linguagem de alto nível para um programa equivalente em código de máquina que é passado para um processador. Geralmente, um compilador não produz diretamente o código de máquina mas sim um programa assembly, semanticamente equivalente ao programa em linguagem de alto nível.

FASES DA COMPILAÇÃO:

- **Análise Léxica;**
- **Análise Sintática;**
- **Análise Semântica;**
- Geração de Código Intermediário;
- Otimização do Código;
- Geração do Código Final.

Análise Léxica

O papel nessa fase de compilação é quebrar as informações de entrada em uma sequência de símbolos léxicos, os **TOKENS**. Por meio desse tratamento serão reconhecidas as palavras reservadas, constantes, identificadores e outros elementos que fazem parte da linguagem de programação em questão.

As **expressões regulares** dão origem a algoritmos de autômatos finito determinísticos e não determinísticos que são utilizados por analisadores léxicos para reconhecer os padrões de cadeias de caracteres.

Análise Sintática

Essa fase depende das informações que o analisador léxico extrai do arquivo fonte. Após o léxico passar os tokens capturados, o sintático utiliza essas informações em paralelo as regras sintáticas com a finalidade de produzir uma árvore sintática.

No caso em questão, a linguagem de programação utilizada como código fonte foi o Mini-Pascal, que possui uma gramática determinística, ou seja, não possui ambiguidade.

Análise Semântica

Com a árvore sintática devidamente elaborada e coerente com a linguagem, a parte semântica percorre a árvore e relaciona os identificadores de acordo com a estrutura hierárquica.

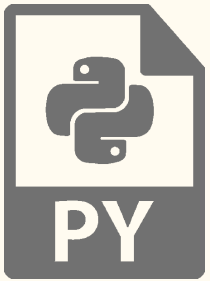
Essa etapa também captura informações sobre o programa fonte para que as fases subsequentes gere o código objeto. Um importante componente dessa fase é a verificação de tipos, nela o compilador verifica se cada operador recebe os operandos permitidos e especificados na linguagem fonte.

Desenvolvimento

Linguagem Utilizada

O python foi a linguagem de programação utilizada para desenvolver os analisadores(léxico, sintático e semântico).

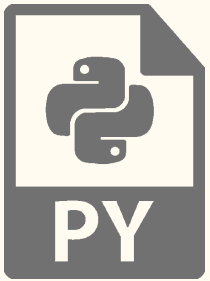
Ela é uma linguagem de alto nível, dinâmica, interpretada, modular, multiplataforma e orientada a objetos. Além disso, possui forma específica de organizar softwares, em que, procedimentos estão submetidos às classes, proporcionando maior controle e estabilidade de código para projetos grandes.



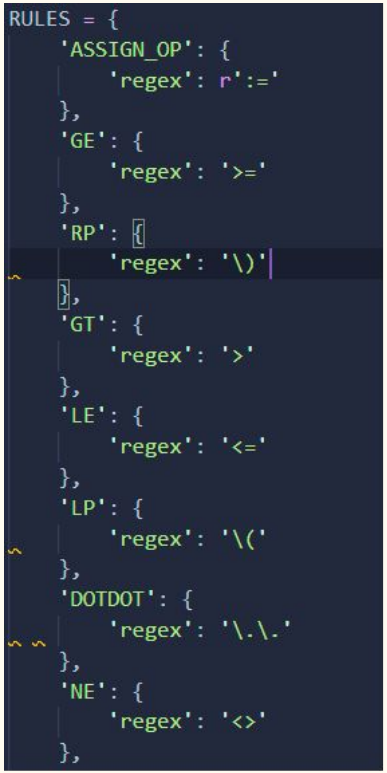
Programação Léxica

Nessa fase, foi de suma importância utilizar a biblioteca **RE**(regex) para a realização de busca de expressões regulares que tem por finalidade retornar resultados baseados no estado de avaliação do programa, ou seja, realizar captura de informações específicas conforme percorre o código fonte.

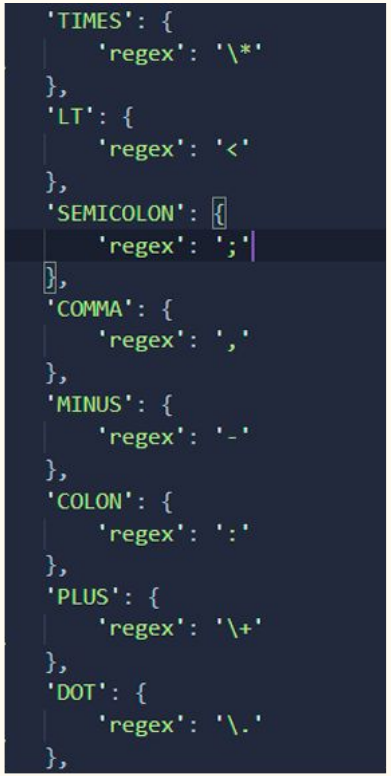
Segue o trecho de código das regras utilizadas em conjunto com a biblioteca REGEX.



Programação Léxica



```
RULES = {
    'ASSIGN_OP': {
        'regex': r':='
    },
    'GE': {
        'regex': '>='
    },
    'RP': {
        'regex': '\)'
    },
    'GT': {
        'regex': '>'
    },
    'LE': {
        'regex': '<='
    },
    'LP': {
        'regex': '\('
    },
    'DOTDOT': {
        'regex': '\.\.'
    },
    'NE': {
        'regex': '<>'
    },
```



```
    'TIMES': {
        'regex': '\*'
    },
    'LT': {
        'regex': '<'
    },
    'SEMICOLON': {
        'regex': ';'
    },
    'COMMA': {
        'regex': ','
    },
    'MINUS': {
        'regex': '-'
    },
    'COLON': {
        'regex': ':'
    },
    'PLUS': {
        'regex': '\+'
    },
    'DOT': {
        'regex': '\.'
    },
```

```
    'LB': {
        'regex': '\['
    },
    'RB': {
        'regex': '\]'
    },
    'EQUAL': {
        'regex': r'='
    },
    'SINGLE_QUOTE': {
        'regex': r'\''
    },
    "DOUBLE_QUOTE": {
        'regex': r'\"'
    },
    'IDENTIFIER': {
        'regex': r'[a-zA-Z][a-zA-Z0-9]*',
    },
    'SINGLE_COMMENT': {
        'regex': r'{.*}',
        'ignore': True,
    },
    'NUM': {
        'regex': r'\d+(?:\d+)*(?:[Ee][+-]?\d+)?',
        'has_attribute': True
    },
```

Programação Léxica

```
'SPACE': {
    'regex': r'[ ]',
    'ignore': True,
},
'TAB': {
    'regex': r'\t',
    'ignore': True,
},
'NEW_LINE': {
    'regex': r'\r??\n',
    'ignore': True,
    'newline': True,
}
```

A chamada RULES, da imagem apresentada, é referente às regras utilizadas na biblioteca REGEX para efetuar buscas de padrões no código fonte, essas regras são tratadas como expressões regulares.

Programação Léxica

Fez necessário criar um método que chamamos de **NEXTTOKEN()**, este tem por finalidade consumir o arquivo sempre que chamado, validando cada token de acordo com as regras(RULES) pré-definidas.

Programação Léxica

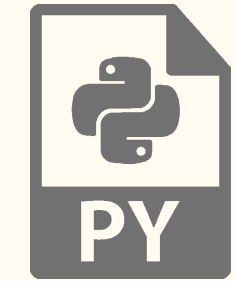
```
def nextToken(self):
    searchToken = None
    if self.endCode():
        return False
    for token in self.rules.keys():
        info = self.rules[token]
        info['compiled'] = re.compile(info['regex'])
        # print(info)
        searchToken = info['compiled'].match(self.code, self.index)

        if searchToken:
            break
    # if token is found
    if searchToken:
        # if new line found, increase CURRENT_LINE
        if "newline" in self.rules[token]:
            Lexico.CURRENT_LINE += 1
            #print(Lexico.CURRENT_LINE)
        # set index where the token found terminate
        self.index += len(searchToken.group())
    else:
        self.index+=1
    return Token(searchToken.group().upper(), token, Lexico.CURRENT_LINE)
```

Trecho de código correspondente.

Programação Sintática

O desenvolvimento deste analisador, demandou a criação de alguns métodos, dentre eles trataremos de três em específico, que são essenciais para se ter um resultado sólido nessa fase, são eles: **EXPECTEDTOKEN()**, **CONSUMESTOKEN()** e **COMPARE()**.

O desenvolvimento desse projeto acarretou na criação de um relatório em que é feita a descrição do código.



Programação Sintática

No caso do **COMPARE()**, têm como função comparar se uma regra ou palavra reservada passada através do parâmetro tk corresponde ao token atual, retornando True caso válido se não False.

```
def compare(self, tk, currentToken):
    reserved = currentToken.getTkName() in self.lexer.tks_reserved
    found = False
    if reserved:
        found = currentToken.getTkName() == tk
    else:
        found = currentToken.getTkType() == tk

    return found
```

Programação Sintática

O método **CONSUMESTOKEN()**, como o nome sugere, tem como função consumir os tokens que o léxico envia; quando falamos em consumir, significa atender à verificação esperada.

```
def consumesToken(self, tk):
    rules = self.lexer.rules

    if self.compare(tk, self.currentToken):
        # self.Arv.tk_OnHear(self.currentToken.getTkName())
        if not self.lexer.endCode():
            self.currentToken = self.lexer.nextToken()
            while 'ignore' in rules[self.currentToken.getTkType()] and rules[self.currentToken.getTkType()]['ignore'] == True:
                self.currentToken = self.lexer.nextToken()
            return True
        else:
            return True
    return False
```

Programação Sintática

Se tratando do método **EXPECTEDTOKEN()** este recebe como parâmetro um termo específico(palavra reservada) ou uma regra(rules) com a finalidade de verificar se o token atual corresponde ao mesmo, para em seguida consumi-lo. A chamada desse método é feita no decorrer da criação das regras de verificação de sintaxe, caso o token não corresponda a regra ou palavra reservada passada como parâmetro é lançada uma exceção de erro sintático.

```
def expectedToken(self, tk):
    if self.consumesToken(tk):
        return True
    else:
        raise ParserException(" Expected "+tk+" but found is '"+self.currentToken.getTkName()+"'", str(self.currentToken.getTkLine()))
        return False
```

Programação Semântica

Após a criação da árvore sintática, essa é utilizada para verificar aspectos relacionados ao significado das instruções, nesse momento ocorre a validação de uma série de regras que não podem ser verificadas nas etapas anteriores.

A análise semântica percorre a árvore sintática relacionando os identificadores com seus dependentes de acordo com a estrutura hierárquica. A verificação de tipos, em que o compilador verifica se cada operador recebe os operandos permitidos e especificados na linguagem fonte é uma fase importante do processo.

Programação Semântica

Uma das funções do Semântico é verificar se um identificador foi definido antes do seu uso. Criamos a função **CHECKVARIABLEDECLARED**() para validar essa regra.

```
def checkVariableDeclaredID(self, tableSimb, token, scope):
    busca = tableSimb['VAR'].get(scope)
    if busca:
        if self.checkToken(busca, token):
            return True
        else:
            raise Exception("Erro Semantico: { Message: IDENTIFIER '"+token.getTkName()+"' undeclared, linha "+str(token.getTkLine())+" }")
    else:
        raise Exception("Erro Semantico: { Message: IDENTIFIER '"+token.getTkName()+"' undeclared, linha "+str(token.getTkLine())+" }")

    return False
```

Programação Semântica

Outro exemplo é a verificação de tipos de variáveis na execução de expressões ou atribuição. A função **CHECKTYPEVARIABEL()** executa esta verificação.

```
def checkTypeVariable(self, tableSimb, variableAssign, token, scope):
    busca = tableSimb['VAR'].get(scope)
    if self.checkVariableDeclaredID(tableSimb, token, scope) and variableAssign:
        if self.type(tableSimb, token, scope) == variableAssign:
            return True
        else:
            raise Exception("Erro Semantico: { Message: IDENTIFIER '"+token.getTkName()+"' type is not valid"+
            " for assignment, linha "+str(token.getTkLine())+" }")
    return False
```

COMPILADORES

Alunos:

- Francisco Carvalho;
- Jorge Luiz;
- Marcos Antonio;
- Vinícius Fontinele.

Universidade
Estadual do Piauí