Material Completo de Aula – Análise Semântica em Compiladores

1. Introdução Conceitual

A análise semântica é uma das fases essenciais do processo de compilação, que sucede a análise sintática e antecede a geração de código intermediário. O seu principal objetivo é garantir que um programa seja logicamente consistente e siga regras específicas que transcendem a correta formação sintática.

Dentre as principais tarefas da análise semântica estão:

- Verificação de compatibilidade entre tipos.
- Verificação de declarações prévias e controle de escopo.
- Gerenciamento e atualização da tabela de símbolos.

---

2. Estruturas Básicas do Processo

Um compilador simples envolve geralmente as seguintes fases principais:

- **Lexer:** Responsável por transformar o código-fonte em tokens.
- **Parser:** Responsável por validar a estrutura sintática do programa segundo uma gramática formal.
- **Analisador Semântico:** Responsável por validar aspectos contextuais como tipos e escopos, com base na árvore sintática ou diretamente integrado ao parser.

---

3. Implementação Detalhada: Exemplo Prático

Neste exemplo, utilizamos Python com uma abordagem didática, implementando as três etapas citadas sem bibliotecas externas:

3.1 Gramática LL(1) Utilizada

```
programa  → stmt_list
stmt_list → stmt stmt_list | ϵ
stmt      → int ID ; | ID = expr ;
expr      → term expr'
expr'     → (+|-) term expr' | ϵ
term      → ID | NUMBER
```

3.2 Descrição das Etapas

- **Lexer:** Implementado manualmente com expressões regulares para geração de tokens.
- **Parser LL(1) Tabular:** Usa uma tabela predefinida de parsing com conjuntos FIRST e FOLLOW pré-calculados.
- **Analisador Semântico:** Integrado ao parser LL(1), com gerenciamento da tabela de símbolos para controle de variáveis declaradas.

---

4. Exemplo

```
import re
```

```
# --- Lexer ---
TOKEN_REGEX = [
    ('INT', r'int\b'),
    ('ID', r'[a-zA-Z_]\w*'),
    ('NUMBER', r'\d+'),
    ('PLUS', r'\+'),
    ('MINUS', r'-'),
    ('EQUALS', r'='),
    ('SEMICOLON', r';'),
    ('WHITESPACE', r'\s+'),
]

class Token:
    def __init__(self, type, value):
        self.type = type
        self.value = value

def lexer(input_code):
    tokens, index = [], 0
    while index < len(input_code):
        match = None
        for token_type, token_regex in TOKEN_REGEX:
            pattern = re.compile(token_regex)
            match = pattern.match(input_code, index)
            if match:
                if token_type != 'WHITESPACE':
                    tokens.append(Token(token_type, match.group(0)))
                index = match.end()
                break
        if not match:
            raise Exception(f'Caractere ilegal "{input_code[index]}" na posição {index}.')
```

```
    tokens.append(Token('$', '$'))  # Fim da entrada
    return tokens

# --- Parser LL(1) Tabular com Análise Semântica ---
class LL1Parser:
    def __init__(self, tokens):
        self.tokens = tokens
        self.pos = 0
        self.stack = ['$', 'programa']
        self.symbol_table = {}

        # Tabela LL(1)
        self.table = {
            ('programa', 'INT'): ['stmt_list'],
            ('programa', 'ID'): ['stmt_list'],
            ('programa', '$'): ['stmt_list'],

            ('stmt_list', 'INT'): ['stmt', 'stmt_list'],
            ('stmt_list', 'ID'): ['stmt', 'stmt_list'],
            ('stmt_list', '$'): [],

            ('stmt', 'INT'): ['INT', 'ID', 'SEMICOLON'],
            ('stmt', 'ID'): ['ID', 'EQUALS', 'expr', 'SEMICOLON'],

            ('expr', 'ID'): ['term', 'expr\''],
            ('expr', 'NUMBER'): ['term', 'expr\''],

            ('expr\'', 'PLUS'): ['PLUS', 'term', 'expr\''],
            ('expr\'', 'MINUS'): ['MINUS', 'term', 'expr\''],
            ('expr\'', 'SEMICOLON'): [],
```

```
            ('term', 'ID'): ['ID'],
            ('term', 'NUMBER'): ['NUMBER'],
        }

    def current_token(self):
        return self.tokens[self.pos]

    def match(self, expected_type):
        token = self.current_token()
        if token.type == expected_type:
            self.pos += 1
        else:
            raise Exception(f'Erro sintático: Esperado "{expected_type}", encontrado "{token.type}".')

    def parse(self):
        while self.stack:
            top = self.stack.pop()
            token = self.current_token()

            if top in ('INT', 'ID', 'NUMBER', 'PLUS', 'MINUS', 'EQUALS', 'SEMICOLON', '$'):
                self.match(top)

            elif top in self.table:
                production = self.table.get((top, token.type))
                if production is None:
                    raise Exception(f'Erro sintático: Sem produção para ({top}, {token.type}).')

                # Expansão dos símbolos da produção
                for symbol in reversed(production):
                    if symbol != 'ϵ':
                        self.stack.append(symbol)
```

```
                # Análise Semântica integrada
                if production == ['INT', 'ID', 'SEMICOLON']:
                    var_name = self.tokens[self.pos + 1].value
                    if var_name in self.symbol_table:
                        raise Exception(f'Erro semântico: variável "{var_name}" já declarada.')
                    self.symbol_table[var_name] = 'int'

                elif production == ['ID', 'EQUALS', 'expr', 'SEMICOLON']:
                    var_name = token.value
                    if var_name not in self.symbol_table:
                        raise Exception(f'Erro semântico: variável "{var_name}" não declarada.')

                elif production == ['ID']:
                    var_name = token.value
                    if var_name not in self.symbol_table:
                        raise Exception(f'Erro semântico: variável "{var_name}" não declarada.')
            else:
                raise Exception(f'Erro sintático inesperado no símbolo "{top}".')

        print('Análise concluída com sucesso.')
        print('Tabela de símbolos:', self.symbol_table)

# --- Código para teste ---
if __name__ == "__main__":
    codigo = '''
        int x;
        int y;
        x = 10;
        y = x + 20;
    '''
```

```
tokens = lexer(codigo)
parser = LL1Parser(tokens)
parser.parse()
```

Saída esperada:

```
Análise concluída com sucesso.
Tabela de símbolos: {'x': 'int', 'y': 'int'}
```

---

5. Discussões Complementares

Sugere-se estimular os alunos a discutirem ou implementarem melhorias como:

- Tratamento mais robusto de erros.
- Inclusão de novos tipos (float, string) na linguagem.
- Tratamento de escopos locais e globais.