Trabalho final de compiladores
Compilador para a Linguagem Fortall

Introdução

Este documento detalha a implementação de um interpretador para a Fortall, uma linguagem de programação imperativa e didática. O projeto abrange as três fases clássicas da compilação: análise léxica, sintática e semântica. A fase final, no entanto, em vez de gerar código de máquina, realiza a interpretação e execução direta do código-fonte, validando sua estrutura e semântica em tempo real.

Guia de Execução

1. Linguagem: python 3 (https://www.python.org/downloads/)
2. Ambiente(opcional):
    a. Criar ambiente: `'python3 -m venv venv'`
    b. Ativar ambiente:
        i. windows: `'source venv/Scripts/activate'` ou
        ii. linux: `'source venv/bin/activate'`
3. Instalar biblioteca: ply `'pip install ply'` (https://www.dabeaz.com/ply/ply.html)
4. Executar: `'python main.py teste.f'`

Gramática da Linguagem Fortall

```
<PROG>          -> programa id; <DECLARACOES> <COMPOSTO> .

<DECLARACOES>   -> <DECLARACAO> <_DECLARACAO> | ε
<DECLARACAO>    -> var <ID> <_ID> : <TIPO> ;
<_DECLARACAO>   -> <DECLARACAO> <_DECLARACAO> | ε
<TIPO>          -> inteiro | logico
<ID>            -> id <_ID>
<_ID>           -> , <ID> <_ID> | ε

<COMPOSTO>      -> inicio <COMANDOS> fim
<COMANDOS>      -> <COMANDO>; <_COMANDO> | ε
<COMANDO>       -> <ATRIBUICAO> | <LEITURA> | <ESCRITA> | <COMPOSTO>
                 | <CONDICIONAL> | <REPETICAO>
<_COMANDO>      -> <COMANDO>; <_COMANDO> | ε
<ATRIBUICAO>    -> id := <EXPR> | id := <EXPR_LOGICA>
<LEITURA>       -> Ler(<ID><_ID>)
<ESCRITA>       -> Escrever (<ESPR_STR>)
<CONDICIONAL>   -> se <EXPR_LOGICA> entao <COMANDOS> senao <COMANDOS> fim
                 | se <EXPR_LOGICA> entao <COMANDOS> fim
<REPETICAO>     -> enquanto <EXPR_LOGICA> faca <COMANDOS> fim

<ESPR_STR>      -> <ITEM_ESCRITA> <_ITEM_ESCRITA> | ε
<ITEM_ESCRITA>  -> <EXPR> | <EXPR_LOGICA> | str | id
<_ITEM_ESCRITA> -> , <ITEM_ESCRITA> <_ITEM_ESCRITA> | ε

<EXPR>          -> <TERMO> <_EXPR>
<_EXPR>         -> + <TERMO> <_EXPR> | - <TERMO> <_EXPR> | ε
<TERMO>         -> <FATOR> <_TERMO>
<_TERMO>        -> * <FATOR> <_TERMO> | / <FATOR> <_TERMO> | ε
<FATOR>         -> - <FATOR> | num | id | ( <EXPR> )

<EXPR_LOGICA>   -> <TERMO_LOGICO> <_EXPR_LOGICA>
<_EXPR_LOGICA>  -> || <TERMO_LOGICO> <_EXPR_LOGICA> | ε
<TERMO_LOGICO>  -> <FATOR_LOGICO> <_TERMO_LOGICO>
<_TERMO_LOGICO> -> && <FATOR_LOGICO> <_TERMO_LOGICO> | ε
<FATOR_LOGICO>  -> ! <FATOR_LOGICO> | <RELACIONAL> | ( <EXPR_LOGICA> )
                 | id | bool
<RELACIONAL>    -> <EXPR_REL> <OP_REL> <EXPR_REL> | <EXPR> <OP_INT> <EXPR>
<EXPR_REL>      -> <EXPR> | <EXPR_LOGICA>
<OP_REL>        -> = | <>
<OP_INT>        -> < | <= | > | >=
```

Detalhes da Implementação

1. **Análise Léxica:**

Abordagem: Uso da biblioteca PLY (`Lex`).

Implementação: A classe `LexicalAnalyzer` define os tokens da linguagem. Cada token, desde palavras-chave (como 'programa', 'se', 'enquanto') até operadores (+, :=, <>) e literais (números, strings), é mapeado a uma expressão regular (regex) que o identifica no código-fonte. Foi implementado um tratamento de erros robusto para capturar caracteres não reconhecidos e estruturas léxicas malformadas, como comentários e strings que não foram devidamente fechados, informando a linha do erro.

2. **Análise Sintática:**

Abordagem: Uso da biblioteca PLY (`Yacc`) para implementar um parser e construção de uma Árvore Sintática Abstrata (AST).

Implementação: A classe `SyntaxAnalyzer` traduz as regras da gramática BNF para funções Python (`p_regra`). Durante a análise, o parser não apenas valida a sequência de tokens, mas também constrói uma Árvore Sintática Abstrata (AST). Cada nó da árvore, uma instância da classe `ASTNode`, representa uma construção da linguagem (uma declaração, uma expressão, um comando condicional, etc.), preservando a estrutura hierárquica do código original. A definição de precedência de operadores foi crucial para garantir que expressões aritméticas e lógicas sejam avaliadas na ordem correta.

3. **Análise Semântica e Execução:**

Abordagem: Implementação de um Interpretador utilizando o padrão de projeto *Visitor*.

Implementação: A classe `SemanticAnalyzer` é responsável por duas tarefas críticas. Primeiro, ela realiza a análise semântica, utilizando uma Tabela de Símbolos para gerenciar variáveis, verificar se foram declaradas antes do uso e checar a compatibilidade de tipos em operações como atribuições. Erros semânticos, como o uso de variáveis não declaradas, são coletados e reportados. Em segundo lugar, a classe prepara o código para a execução. Ela transforma a AST em uma nova árvore de "nós semânticos" (`SemanticNode`), onde cada nó contém uma função (`action`) que define seu comportamento. Ao final, uma chamada a esta árvore executa o programa, interpretando cada comando e expressão recursivamente.