UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Câmpus de Rio Claro

Documentação de projeto mybc

Compiladores

Equipe:

André Luis Dias Nogueira
Lucas Abdala Martins
Vitor Marchini Rolisola

Conteúdo

1 Introdução

Esta documentação descreve a calculadora interativa desenvolvida na disciplina de Compiladores, enfatizando a ligação entre as entregas do projeto e os tópicos apresentados em aula. A implementação foi baseada nos materiais de aula, especialmente os roteiros das aulas 3, 4, 5 e 10, que apresentam diagrama sintático, gramáticas em EBNF, construção de autômatos e evolução do parser.

Resumo de Funcionalidades

- Avaliação de expressões aritméticas infixas com ‘+‘, ‘-‘, ‘*‘, ‘/‘.
- Literais: decimal, octal (‘0‘ prefixado), hexadecimal (‘0x‘/‘0X‘), ponto flutuante (inclui notação exponencial), numerais romanos clássicos com notação subtrativa.
- Atribuição com operador ‘:=‘ e variável especial ‘ans‘ para último resultado.
- Tratamento de erros com linha e coluna, recuperação básica de fluxo.
- Conversão interna de numerais romanos via rotina dedicada (‘rmntoi‘).
- Comandos de saída: ‘exit‘, ‘quit‘, ‘q‘.
- Empacotamento automático de fontes via alvo ‘make compact‘ (gera ‘.tar.gz‘).

2 Metodologia e Referencial

O desenvolvimento seguiu um processo incremental:

1. **Modelagem**: com base nos diagramas sintáticos estudados em aula, definimos uma gramática LL(1) para expressões aritméticas estendidas.

2. **Projeto dos Autômatos**: os autômatos determinísticos descritos em `aula 3` orientaram a criação de reconhecedores independentes para identificadores, números em diferentes bases e comandos reservados.

3. **Integração com Ações Semânticas**: a partir das aulas sobre EBNF e pilhas (`aula 4` e `aula 5`), anexamos ações semânticas ao diagrama sintático para gerar código em ordem correta.

4. **Refinamento do Parser**: as versões evolutivas em `aula 10` serviram de base para ajustar a estrutura recursiva, o controle de erros e a tabela de símbolos.

3 Fundamentos Teóricos

3.1 Gramática e EBNF

A gramática adotada é equivalente à apresentada em aula, com níveis para *expressão*, *termo* e *fator*. Em EBNF simplificada:

$$\langle Expressao \rangle ::= [+ \mid -] \langle Termo \rangle \{+ \langle Termo \rangle \mid - \langle Termo \rangle\}$$
$$\langle Termo \rangle ::= \langle Fator \rangle \{* \langle Fator \rangle \mid / \langle Fator \rangle\}$$
$$\langle Fator \rangle ::= (\langle Expressao \rangle) \mid \langle Literal \rangle \mid \langle Identificador \rangle [:= \langle Expressao \rangle]$$

Os literais contemplam bases decimal, octal, hexadecimal, ponto flutuante, expoente e numerais romanos, como discutido em sala.

3.2 Diagramas Sintáticos

Os diagramas sintáticos das aulas foram utilizados diretamente para guiar a recursão do parser. Cada macro no arquivo `parser.c` (`T_BEGIN`/`T_END`, `E_BEGIN`/`E_END`) representa laços equivalentes aos caminhos do diagrama, garantindo a mesma ordem de visitação que as transparências de aula sugerem.

3.3 Autômatos e Reconhecimento Léxico

Seguindo o material de `aula 3`, cada token é reconhecido por um autômato determinístico separado, implementado em funções como `isID`, `isNUM`, `isHEX` e `isROMAN`. O encadeamento dos autômatos na função `gettoken` garante que ambiguidades sejam resolvidas conforme a prioridade definida na especificação da linguagem.

3.4 Recursão e Padrão LL(1)

O parser segue padrão LL(1), lendo tokens da esquerda para a direita e construindo derivações mais à esquerda por meio de chamadas recursivas. O primeiro token é adiantado em `main.c`, e cada chamada de `expression`, `cmd` ou `match` garante que a condição LL(1) seja preservada.

3.5 Tabela de Símbolos e Pilha

Inspirados nas práticas das aulas sobre ambientes de execução, a implementação emprega:

- **Tabela de símbolos linear**: arrays `symtab` e `vmem` mapeiam identificadores para valores, conforme as exercitações de `aula 5`.
- **Pilha de avaliação**: o vetor `stack` e o ponteiro `sp` reproduzem a pilha utilizada para armazenar operandos intermediários de termos e expressões, como indicado nos exemplos de ações semânticas em diagrama sintático.

4 Arquitetura da Implementação

4.1 `main.c`

Responsável por configurar entradas e saídas, registrar o tratador de sinais e iniciar o analisador. O método `handle_sigint` referencia a rotina de controle de interrupções proposta nas aulas sobre integração com o terminal.

4.2 `lexer.c` e `lexer.h`

Os autômatos são estruturados em funções separadas, cada uma documentada com a expressão regular equivalente. A função `skipspaces` implementa o tratamento de espaços, tabulações e sequências de escape, além de manter `line` e `column`, como exigido para relatórios de erro.

4.3 `parser.c` e `parser.h`

O arquivo define:

- o laço principal `mybc`, que consome comandos até receber EOF ou ordens de saída;
- `cmd`, que despacha tokens para expressões ou encerra o programa;
- `expression`, que aplica negacão, soma/subtração e multiplicção/divisão usando pilha e ações semânticas numeradas (as mesmas presentes nos slides de aula sobre diagramas sintáticos);
- funções auxiliares `match`, `recall`, `store` e `rmntoi`, conectando a gramática ao ambiente de execução.

4.4 `tokens.h` e `main.h`

Centralizam a enumeração de tokens e a declaração compartilhada de variáveis e funções. A divisão segue o padrão modular discutido na primeira parte do curso.

4.5 Fluxo de Execução

1. O usuário insere uma expressão; `gettoken` identifica o primeiro token.
2. `cmd` avalia se o token é um comando de saída ou o início de uma expressão.
3. `expression` aplica as regras da gramática, empilhando resultados parciais e executando as ações semânticas.
4. O resultado final é impresso e armazenado em `ans`; variáveis são salvas via `store`.
5. Em caso de erro, `match` e `skipspaces` relatam linha e coluna, recuperando o fluxo conforme os exemplos em aula.

4.6 Exemplos de Uso

A Listagem 4.1 demonstra recursos discutidos em sala, como atribuição, bases numéricas e reutilização de resultados anteriores.

Listing 4.1: Sessão de execução

```
$ ./mybc
(2 + 3) * 4;
20
x := 0x10 + 8;
24
MD;             # numeral romano (M=1000, D=500)
1500
y := ans / II; # uso de divisão com romano
750
ans + 1;
751
quit
```

5 Empacotamento e Release

O projeto inclui um fluxo simples de empacotamento:

1. **Build**: ‘make mybc‘ compila o executável.
2. **Compactação**: ‘make compact‘ gera arquivo ‘mybc_grp9_YYYYMMDD.tar.gz‘ com fontes e Makefile.
3. **Tag opcional**: em repositório Git, criar tag anotada (‘git tag -a v1.0.0 -m "Release v1.0.0"‘).
4. **Publicação**: anexar o ‘.tar.gz‘ à página de releases (GitHub) ou distribuir internamente.

Para incluir coautores em commits, utilizar o trailer ‘Co-authored-by:‘ nas mensagens, conforme recomendação da plataforma de hospedagem.

6 Correções Recentes

- **Lexer de Romanos**: correção de loops que atribuíam resultado booleano de comparação a ‘lexeme[i]‘ (parênteses ausentes), causando perda de caracteres e avaliação incorreta (‘MD‘ retornava 1000). Agora numerais são capturados integralmente e convertidos corretamente (e.g., ‘MD‘ 1500, ‘CDXLIV‘ 444).
- **Documentação**: adicionada seção de release e resumo de funcionalidades.
- **Exemplos**: sessão atualizada demonstrando uso de ‘ans‘ e numerais romanos.

7 Testes e Garantia de Qualidade

Os planos de teste revisitam os cenários propostos no material de apoio:

- expressões aritméticas simples e compostas;
- combinação de bases numéricas e uso de numerais romanos;
- atribuições aninhadas e recuperação de valores com `ans`;
- interrupções com `Ctrl+C` e edições interativas com setas, para validar a limpeza do estado.

8 Conclusão

O projeto sintetiza os conceitos de gramáticas LL(1), autômatos, diagramas sintáticos, ações semânticas, pilhas e tabelas de símbolos estudados em aula. As evoluções recentes reforçam a importância de testes direcionados para validar a captura léxica de padrões e manter a consistência da avaliação semântica.