INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA DO PIAUÍ

CAMPUS TERESINA CENTRAL

TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

GERMANO BARBOSA DA SILVA JÚNIOR

ÁBACO: um interpretador JIT de Visual Basic 6 para a modernização de sistemas legado

TERESINA, PIAUÍ

2026

Germano Barbosa da Silva Júnior

ÁBACO: um interpretador JIT de Visual Basic 6 para a modernização de sistemas legado

Trabalho de Conclusão de Curso (monografia) apresentado como exigência parcial para obtenção do diploma do Curso de Análise e Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia do Piauí, Campus Teresina.

Orientador: Prof. M^{e}. Otílio Paulo

TERESINA, PIAUÍ

2026

Dados Internacionais de Catalogação na Publicação (CIP) de acordo com ISBD

Silva Júnior, Germano Barbosa da

S581a Ábaco : um interpretador JIT de Visual Basic 6 para a mobilização de sistemas legado / Germano Barbosa da Silva Júnior. - 2026.

34 f.: il. color.

Trabalho de Conclusão de Curso (Tecnologia em Análise e Desenvolvimento de Sistemas) - Instituto Federal de Educação, Ciência e Tecnologia do Piauí, Campus Teresina Central, 2026.

Orientador : Prof Me. Otílio Paulo da Silva Neto.

1. Sistemas legado. 2. JIT. 3. Interpretadores. 4. Compiladores. I. Título.

CDD - 004.21

Elaborado por Maria Rosismar Farias CRB CRB 3/631

GERMANO BARBOSA DA SILVA JÚNIOR

ÁBACO: um interpretador JIT de Visual Basic 6 para a modernização de sistemas legado

Trabalho de Conclusão de Curso (monografia) apresentado como exigência parcial para obtenção do diploma do Curso de Análise e Desenvolvimento de Sistemas do Instituto Federal de Educação, Ciência e Tecnologia do Piauí, Campus Teresina.

Aprovada em 24 de Junho de 2026.

BANCA EXAMINADORA:

Prof. M^{e}. Otílio Paulo(Orientador)
Instituto Federal do Piauí (IFPI)

Prof. M^{e}. Duany Dreyton Bezerra Sousa
Instituto Federal do Piauí (IFPI)

Prof. M^{e}. Rogério Da Silva Batista
Instituto Federal do Piauí (IFPI)

TERESINA, PIAUÍ
2026

Dedico este trabalho a todos que acreditaram que ele sairia.

AGRADECIMENTOS

Agradeço a todos os professores e servidores do IFPI do Campus Corrente, pois todos contribuíram direta ou indiretamente para a minha formação acadêmica e profissional. Agradeço especialmente ao meu orientador, que deu todo o apoio necessário para que chegasse até aqui. Agradeço ao meu Deus, que me deu força quando mais precisava. Agradeço a minha família pelo suporte e amor, especialmente ao meu pai por ter me colocado no rumo que tomo hoje.

Nós escolhemos ir para a lua nessa década e fazer as outras coisas,
não porque elas são fáceis, mas porque são difíceis.
John F. Kennedy

RESUMO

Diversas organizações brasileiras ainda dependem de sistemas legados desenvolvidos em linguagens cujos ecossistemas tornaram-se obsoletos, como o Visual Basic 6.0 (VB6). Lançada em 1998 e sem suporte oficial desde 2008, a manutenção dessa tecnologia em ambientes modernos é severamente limitada pela natureza proprietária de seu compilador e pela impossibilidade de aquisição de novas licenças. Este trabalho apresenta o desenvolvimento do Ábaco, um interpretador de código aberto de VB6 escrito em Rust, projetado para facilitar a modernização desses sistemas. Diferente do ambiente original, o Ábaco visa estabelecer uma fundação técnica que viabiliza a futura implementação de capacidades modernas, como execução paralela e integração flexível com bibliotecas externas, enquanto oferece compatibilidade com o legado. A metodologia de implementação compreende todo o pipeline de execução - carregamento, análise sintática e montagem, por meio de compilação JIT (bem na hora) para a arquitetura x86 32-bit - sem dependências externas ao sistema operacional. Os resultados demonstram que o sistema já é capaz de executar um subconjunto da linguagem VB6, estabelecendo uma fundação sólida para a compatibilidade total e para a extensão de funcionalidades em sistemas legados.

Palavras-chaves: sistemas legado; JIT; interpretadores; compiladores.

ABSTRACT

Numerous Brazilian organizations still depend on legacy systems developed in languages whose ecosystems have become obsolete, such as Visual Basic 6.0 (VB6). Released in 1998 and without official support since 2008, the maintenance of this technology in modern environments is severely limited by the proprietary nature of its compiler and the impossibility of acquiring new licenses. This work presents the development of Ábaco, an open-source VB6 interpreter written in Rust, designed to facilitate the modernization of these systems. Unlike the original environment, Ábaco aims to establish a technical foundation that enables the future implementation of modern features, such as parallel execution and flexible integration with external libraries, while offering compatibility with legacy codebases. The implementation methodology covers the entire execution pipeline - loading, syntactic analysis, and assembly, through Just-In-Time (JIT) compilation for the 32-bit x86 architecture - with no external dependencies beyond the operating system. The results demonstrate that the system is already capable of executing a subset of the VB6 language, establishing a solid foundation for total compatibility and the extension of functionalities in legacy systems.

Keywords: legacy systems; JIT; interpreters; compilers.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

IFPI	Instituto Federal de Educação, Ciência e Tecnologia do Piauí
API	Interface de programação de aplicações
ABI	Interface binário de aplicações
UDT	Tipo definido pelo usuário
JIT	Bem na hora
CPU	Unidade central de processamento
GDI	Interface do dispositivo gráfico
ASCII	Código americano padrão para troca de informações

SUMÁRIO

1 INTRODUÇÃO

A linguagem de programação Visual Basic 6.0 (VB6) continua sendo usada amplamente, em aplicações que continuam sendo atualizadas para responder às mudanças no mercado. No entanto, por ter sido criada muito antes de surgirem certas demandas modernas, ela não foi projetada para lidar com a popularização de CPUs com múltiplos núcleos ou com o compartilhamento de bibliotecas implementadas em várias linguagens que existem fora do ecossistema da Microsoft.

Dado que a linguagem e suas ferramentas oficiais já tiveram seu suporte encerrado pela Microsoft em 2008, e que é proprietária, não há caminhos bons para eliminar as limitações da linguagem.

Diversos esforços já foram realizados no sentido de suceder ou substituir a linguagem Visual Basic 6 (VB6), cada qual com abordagens e objetivos distintos. A seguir, são analisados os mais representativos, destacando suas limitações em relação aos requisitos de compatibilidade total com VB6, código aberto e capacidade de gerar executáveis independentes.

O FreeBasic (2026) é um compilador de código aberto para a família BASIC, que gera código de máquina nativo para várias plataformas (Windows, Linux, DOS). Embora suporte muitas construções da linguagem BASIC, o projeto não visa compatibilidade com VB6. Especificamente, o FreeBasic não implementa formulários no modelo COM (Component Object Model) utilizado pelo VB6, nem oferece suporte nativo aos controles ActiveX e à API de automação da Microsoft. Consequentemente, aplicações VB6 que dependem de formulários (`.frm`) e componentes COM não podem ser compiladas diretamente.

A Microsoft introduziu o VB.NET como sucessor do VB6, integrado ao ecossistema .NET. Entretanto, a mudança foi profunda: VB.NET não é compatível com VB6 em nível binário nem sintático completo. Embora existam ferramentas de conversão automática de código (como o assistente de atualização da própria Microsoft e soluções de terceiros), essas ferramentas geralmente exigem reescrita de partes significativas do código, especialmente no que tange a formulários, tratamento de eventos e chamadas à API Windows. O VB.NET gerenciado substitui o modelo COM por assemblies .NET, tornando inviável a migração direta de sistemas legados complexos sem retrabalho

substancial. O trabalho de Devi, Perumal & Mouli (2011) mostra as dificuldades que se pode esperar durante a migração, e propõe meios para tratá-las, mas demonstra como reestruturações são inevitáveis.

O VBA, como definida em MS-VBAL... (2025), é uma sub-linguagem do VB6 embutida em aplicações do Microsoft Office (Excel, Access, Word). Embora compartilhe a mesma sintaxe e o mesmo modelo de objetos COM, o VBA não foi projetado para gerar executáveis independentes (.exe). Seu propósito é automação de documentos. Sistemas legados em VB6 que precisam ser distribuídos como aplicações autônomas não podem ser executados em VBA. Além disso, o ambiente de execução do VBA é fortemente acoplado ao aplicativo hospedeiro, o que inviabiliza seu uso para sistemas de missão crítica fora do ecossistema Office.

O RadBasic (2026) é um compilador comercial para VB6, que promete compatibilidade total com a linguagem original, suporte a formulários, compilação para código nativo e geração de executáveis. Entretanto, trata-se de um produto proprietário e pago, o que limita seu uso por projetos acadêmicos, de código aberto ou por empresas com restrições orçamentárias. Além disso, seu código-fonte não está disponível para auditoria ou extensão.

Assim como o RadBasic (2026), o TwinBasic (2026) também se trata de um compilador comercial, que se propõe a entregar compatibilidade total com o VB6. Apesar de ter uma edição de comunidade com boa compatibilidade, várias funcionalidades estão restritas sob uma inscrição. O trabalho de Ala-Hulkko (2025) concluiu que a conversão da aplicação crítica em VB6 para TwinBasic (2026) é a melhor abordagem para a empresa sujeita do estudo.

A análise de Dalle & Jullien (2001) mostra como a disponibilidade de uma versão gratuita não substitui a necessidade de publicar o código fonte, pois o código aberto possibilita auditorias e extensões sem intervenções do desenvolvedor original, e confere confiança aos usuários de que seus projetos não se tornarão incompiláveis repentinamente, pois a empresa deixou de dar suporte ou desligou o produto.

Para preencher essa lacuna, este trabalho apresenta o Ábaco, um interpretador que se distingue por três características combinadas:

1. **Compatibilidade com VB6**: suporta formulários, eventos, GDI e chamadas a APIs Windows, conforme demonstrado pelo teste de integração (Seção 3.5.5).

2. **Código fonte aberto**: desenvolvido em Rust, com licença GPLv3, permitindo auditoria, modificação e redistribuição.

3. **Compilação JIT**: gera código de máquina nativo dinamicamente, com tempos de compilação baixos.

Até o momento, não foi identificado na literatura outro compilador JIT de código aberto que atenda simultaneamente ou se proponha a atender a esses três requisitos.

O Ábaco precisa ser capaz de interpretar código VB6 da mesma forma que o original, para que a execução se mantenha fiel ao comportamento antigo. Precisa também possibilitar que novas capacidades sejam adicionadas, além das capacidades originais. Visando a viabilização da modernização de projetos legados, além da sua manutenção.

Para realizar isso, foi utilizada uma abordagem estruturada em múltiplas etapas, onde a saída da etapa anterior é usada pela subsequente. A saída da última etapa é código armazenado em memória executável, o que reduz a execução do código do usuário à chamada da função `main`. Para validar e avaliar a implementação, foi utilizado um teste de integração, uma bateria de testes unitários e benchmarks medindo vários aspectos da linguagem.

Dado isso, o Ábaco estabelece uma fundação para a criação de uma ferramenta livre para o desenvolvimento e modernização de soluções em VB6.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 VB6 (VISUAL BASIC 6)

O Visual Basic 6 é uma linguagem de programação orientada a eventos e baseada em objetos, cujo modelo de objetos fundamenta-se no COM (Modelo Componente Objeto). A API de automação (`IDispatch`) é amplamente utilizada para acessar propriedades e métodos de objetos em tempo de execução.

2.1.1 RAD e ciclo de desenvolvimento

O VB6 foi concebido para aplicação de desenvolvimento rápida (RAD), permitindo que desenvolvedores criassem interfaces gráficas com arrastar-e-soltar e escrevessem lógica de negócio rapidamente. Essa abordagem favorece ciclos de iteração curtos, nos quais a edição de código e a execução do programa se alternam com frequência. O interpretador Ábaco mantém essa característica ao apresentar tempos de compilação muito baixos (Tabela **??**). Uma característica notável do ambiente VB6 é a capacidade de editar código enquanto o programa está em execução, conhecida hoje como *hot-reloading*. O interpretador Ábaco não implementa essa funcionalidade, mas a implementação atual possibilita que trabalhos futuros adicionem essa capacidade.

2.1.2 Tratamento de exceções

O VB6 implementa tratamento de exceções não estruturadas através das construções `On Error GoTo` e `Resume`. O que impõe um custo de execução, pois cada função deve manter uma estrutura de exceção na pilha, mesmo quando nenhum erro ocorre.

2.1.3 P-Code e compilação nativa

Como evidenciado por Chubchenko (2025), o VB6 pode executar código de duas formas:

- **Modo interpretado (P-Code)**: o código fonte é compilado para um *bytecode* proprietário chamado P-Code (Pseudo-Código), que é interpretado pela biblioteca

`msvbvm60.dll`. Esse modo oferece compilação rápida e depuração mais rica, mas com desempenho inferior.

- **Compilação nativa**: o VB6 pode gerar código de máquina diretamente para a arquitetura x86, produzindo executáveis convencionais. Contudo, mesmo nesse modo, partes do ambiente de execução (como a biblioteca de runtime) ainda dependem da DLL `msvbvm60.dll`.

O interpretador Ábaco não utiliza P-Code; ele traduz o código fonte diretamente para código de máquina x86 de 32 bits, eliminando a camada de interpretação.

2.2 INTERPRETADORES, COMPILADORES E COMPILAÇÃO JIT (BEM NA HORA)

2.2.1 Definições

Segundo o livro de Alfred *et al.* (2007), é possível entender a diferença entre interpretadores e compilador da seguinte forma:

Um **interpretador** é um programa que recebe como entrada um código fonte (ou *bytecode*) e produz os efeitos especificados por esse código, sem gerar um artefato executável separado. O interpretador analisa e executa cada comando sequencialmente.

Um **compilador** é um programa que traduz o código fonte para um artefato executável (como um arquivo `.exe` ou biblioteca), que pode ser armazenado em disco e executado posteriormente sem a presença do compilador. A separação entre tempo de compilação e tempo de execução é estrita.

2.2.2 Comparação entre interpretação e compilação

- **Desempenho**: compiladores geralmente produzem código mais rápido, pois as otimizações são aplicadas uma única vez e a execução não tem *overhead* de interpretação. Interpretadores podem ser mais lentos devido à decodificação de instruções a cada iteração.

- **Portabilidade**: código interpretado tende a ser mais portável (desde que o interpretador exista para a plataforma), enquanto código compilado é específico da arquitetura.

- **Tempo de desenvolvimento**: interpretadores oferecem ciclos de edição-execução mais curtos (compilação ausente ou muito rápida), favorecendo o desenvolvimento iterativo.

2.2.3 Compilação Just-In-Time (JIT)

O JIT é uma técnica que combina aspectos de interpretação e compilação. Um compilador JIT traduz partes do código (métodos, funções, traços) para código de máquina durante a execução do programa. Os principais tipos de JIT são:

- **JIT *eager***: todo o código é compilado antes da execução.
- **JIT *lazy***: a compilação ocorre apenas quando uma função é chamada pela primeira vez.
- **JIT *tracing***: registra traços de execução e compila apenas os caminhos frequentemente percorridos.
- **Compilação adaptativa (híbrida)**: combina interpretação inicial com compilação de métodos "quentes"(ex: Java HotSpot).

2.2.4 Exemplos de compiladores JIT

- V8... (2026): compilador JIT para JavaScript, usado no Google Chrome e NodeJS, usa JIT tracing (ignition + turbofan).
- Kotzmann *et al.* (2008): compilação adaptativa para bytecode Java.
- SpiderMonkey... (2026): JIT para JavaScript, usado no Mozilla.
- LuaJIT (2025): compilador JIT de alto desempenho para Lua, notável por gerar código muito eficiente.

O Ábaco inspira-se no modelo *eager* JIT, buscando um equilíbrio entre simplicidade, baixo *overhead* de inicialização e boa performance.

2.3 ALGORITMOS

2.3.1 Algoritmo Branco-Cinza-Preto para detecção de ciclos

Em problemas de análise de dependências, como a avaliação de constantes e a determinação de layout de tipos definidos pelo usuário, é necessário detectar ciclos que tornam o programa inválido. Um algoritmo clássico para detecção de ciclos em grafos direcionados é a variação da busca em profundidade (DFS), como descrito no livro de Cormen *et al.* (2022), que utiliza três cores para cada vértice:

- **Branco**: vértice não visitado.
- **Cinza**: vértice em visita (na pilha de recursão atual).
- **Preto**: vértice totalmente explorado.

Se durante a DFS um arco leva a um vértice ainda cinza, um ciclo é detectado. O interpretador Ábaco utiliza essa estratégia tanto para constantes (Seção 3.3.3) quanto para tipos definidos pelo usuário (Seção 3.3.4).

2.3.2 Algoritmo Shunting-yard de Dijkstra

A conversão de expressões infixas (como `a + b * c`) para notação pós-fixa (ou notação polonesa reversa) é uma etapa comum na compilação. O algoritmo Shunting-yard (pátio de manobras), proposto por Dijkstra (1961), processa tokens da expressão da esquerda para a direita, utilizando uma pilha de operadores e uma fila de saída. As regras de precedência e associatividade determinam quando operadores são desempilhados. Modificações podem ser introduzidas para tratar chamadas de função, operadores unários e parênteses.

No interpretador Ábaco, uma versão modificada do Shunting-yard é utilizada para gerar uma sequência pós-fixa que é então traduzida para instruções nativas x86. As modificações incluem verificação de sintaxe e análise além de operadores infixo, como tratamento de operadores prefixos (como `Not` e `+/-` unário) e chamada de função.

2.4 CONVENÇÕES DE CHAMADA, LAYOUT DE ESTRUTURAS E A ARQUITETURA X86

2.4.1 Arquitetura x86 de 32 bits

A arquitetura Intel x86 no modo de 32 bits (IA-32), como definida em Intel®... (2026), é uma especificação de instruções de código de máquina, que define como uma CPU deve se comportar ao executar cada instrução. Notavelmente, possui registradores de propósito geral (`EAX`, `EBX`, `ECX`, `EDX`, `ESI`, `EDI`, `EBP`, `ESP`), além de registradores de segmento e de controle. A pilha (*stack*) cresce em direção a endereços decrescentes, com `ESP` apontando para o topo. As instruções precisam ser codificadas na forma de bytes para que a CPU possa executá-los, assim como especificado no manual para a arquitetura x86 da Intel.

2.4.2 Convenções de chamada

Uma convenção de chamada define como argumentos são passados, quem limpa a pilha e quais registradores devem ser preservados. Segundo o artigo Calling... (2021), as principais são:

- **stdcall**: os argumentos são empilhados da direita para a esquerda; o chamado (*callee*) limpa a pilha. É a convenção padrão da API Win32 e a usada pelo VB6 para chamadas a funções externas (declaração `Declare Function`).
- **cdecl**: usada em C; o chamador limpa a pilha. Embora o VB6 possua a palavra chave `CDecl` reservada, não há suporte para essa convenção, dificultando a integração com bibliotecas que usam essa convenção.

O interpretador Ábaco adota `stdcall` para todas as funções, compatível com o runtime do VB6 e com as chamadas a APIs.

2.5 PROTEÇÃO DE MEMÓRIA

Para que código de máquina gerado dinamicamente possa ser executado, é necessário que a região de memória que o contém tenha permissão de execução. Seguindo as definições de Intel®... (2026), CPUs implementam a proteção de acesso em páginas de memória virtual. Sistemas operacionais modernos, como Windows,

adotam essa proteção de memória baseada em páginas, com permissões separadas para leitura, escrita e execução.

No Windows, a função `VirtualAlloc` permite alocar memória com permissões personalizadas. Para gerar código JIT, é comum alocar uma página com permissões `PAGE_READWRITE`, escrever as instruções e, em seguida, alterar a proteção para `PAGE_EXECUTE_READ` (ou `PAGE_EXECUTE_READWRITE` se a regravabilidade for necessária). O interpretador Ábaco utiliza `VirtualAlloc` com `PAGE_EXECUTE_READWRITE`.

Além disso, o Windows mantém um cache de instruções da CPU. Após escrever código de máquina na memória, é necessário chamar `FlushInstructionCache` para invalidar o cache e garantir que a CPU execute a versão mais recente. O Ábaco faz essa chamada antes de transferir o controle para o código gerado.

Por fim, não é possível simplesmente pular para um endereço de memória comum que não tenha permissão de execução; o sistema operacional levantaria uma exceção de acesso violação. Assim, o uso de APIs de alocação executável é fundamental para qualquer compilador JIT.

3 METODOLOGIA

3.1 AMBIENTE TÉCNICO

Para a implementação, foi escolhida a linguagem de programação Rust, versão 1.92.0, por permitir acesso às APIs do sistema operacional, pela fluidez no controle de aspectos de baixo nível, pelas restrições de mutabilidade que auxiliam na prevenção de problemas e pela qualidade do código de máquina gerado. O programa foi desenvolvido no sistema operacional Windows 11 IoT Enterprise LTSC, versão 10.0.26100, na arquitetura x86 de 64 bits.

Dado o profundo envolvimento do VB6 com o sistema operacional Windows, especificamente na arquitetura x86 de 32 bits, optou-se por suportar apenas esse sistema operacional e essa arquitetura. Isso foi realizado usando o *toolchain* `i686-pc-windows-msvc`.

No desenvolvimento, foi utilizado o pacote *windows-sys*, versão 0.61.2, para viabilizar o acesso às APIs do sistema operacional.

3.2 GRAMÁTICA

O código VB6 é codificado com Windows-1252, que é uma codificação legada de um byte por caractere. Por isso, não é possível escrever no código caracteres não representáveis nessa codificação, como emojis.

A sintaxe do código VB6 é baseada em linhas, nas quais cada linha tipicamente pode conter no máximo um comando.

Por sua vez, as linhas são compostas por sequências de tokens.

3.2.1 Tokens

Tokens podem ser classificados em identificadores, palavras-chave, símbolos de pontuação e valores literais.

Identificadores são palavras que podem servir para nomear declarações do usuário, como rotinas ou variáveis. Eles consistem em uma sequência de um ou mais letras, dígitos ou *underscore*, não são iniciados por um dígito, são opcionalmente terminados por um caractere sufixo de tipo e não são uma palavra-chave.

Um exemplo de identificador é `cliente`.

Palavras-chave são identificadores reservados pela linguagem para indicar semânticas especiais e não podem ser usadas para nomear declarações. Um exemplo de palavra-chave é `function`.

Os símbolos de pontuação são um subconjunto dos caracteres ASCII imprimíveis que não são letras, números ou caracteres de controle. Assim como palavras-chave, os símbolos de pontuação têm significado especial, único para cada símbolo. Um exemplo de símbolo de pontuação é '&', que é responsável pela concatenação de *strings*.

Finalmente, valores literais são tokens que representam um valor específico, de um certo tipo.

Algumas palavras chaves tem função de valor literal como `True`, que é tanto valor literal quanto palavra-chave. Fora palavras chaves, texto delimitado por aspas duplas constitui um valor literal *string*. Tokens que começam com um dígito ASCII, ou com a sequência `&H` e são compostos por dígitos ASCII são valores literais numéricos, decimais e hexadecimais respectivamente. E podem opcionalmente ter um caractere sufixo de tipo no fim.

3.2.2 Infraestrutura comum

Como consequência da abordagem de análise sob demanda, todas as etapas realizam análise léxica, sintática e semântica. Somado a isso, as etapas necessitam que certos serviços estejam disponíveis, como a busca de declarações por nome ou a emissão de erros de compilação. Para evitar duplicação de código, componentes compartilhados foram desenvolvidos para auxiliar na análise do código do usuário.

3.2.3 Análise léxica

Foram implementadas funções que consomem parte do código-fonte da esquerda para a direita. Cada função procura por um token ou conjunto de tokens específico. Ao executar, essas funções verificam se o restante do código-fonte começa com o token ou tokens desejado. Em caso afirmativo, removem do código-fonte os tokens e espaços em branco que os seguem, retornando o que foi consumido com sucesso. Caso o restante do código-fonte não comece com o token desejado, o código-fonte não é consumido e um erro de compilação é retornado.

A consequência é que se torna possível chamar essas funções em sequência,

condicionalmente e repetidamente, para consumir estruturas de código mais complexas do que tokens individuais.

Essas funções foram: `take_ident`, que consome um identificador e o retorna em caso de sucesso; `take_word_opt`, que consome a palavra-chave ou símbolo de pontuação fornecido e retorna verdadeiro em caso de sucesso; `take_string`, que consome um valor literal *string* e o retorna em caso de sucesso; `take_expr`, que consome uma expressão e a retorna em caso de sucesso; `take_lhs`, que consome uma expressão que fica a esquerda de uma atribuição ou chamada e a retorna em caso de sucesso, expressões desse tipo requerem análise especial para não consumir erroneamente tokens subsequentes.

Utilizando apenas essas funções, é possível realizar a análise sintática de todas as linhas suportadas pela implementação.

3.2.4 Linhas e sua análise sintática

Linhas são compostas por uma sequência possivelmente vazia de tokens. No anexo A há uma lista de todas as linhas suportadas pela implementação atual da análise sintática de linhas.

Em uma linguagem na qual os comandos naturalmente se dividem em linhas, a análise sintática consiste em interpretar a sequência de tokens de uma linha, decidir qual comando está sendo analisado e extrair suas informações.

3.2.5 Expressões

Expressões são porções de código que, quando avaliadas, produzem um valor de um determinado tipo. Exemplos incluem operações aritméticas (como `1 + 2` ou `x * 3`), comparações lógicas (`a > b`), chamadas de função (`MeuMétodo(10)`) e literais. A gramática de expressões do VB6 é rica, incluindo operadores aritméticos, relacionais, lógicos e de concatenação.

Os operadores suportados na implementação atual são listados na Tabela **??**, com suas precedências.

Tabela 1 – Operadores suportados no compilador Ábaco

Tipo	Operadores	Precedência
Aritméticos	^ (exponenciação)	1
	− (unário), + (unário)	2
	*	3
	/	4
	\ (divisão inteira)	5
	Mod	6
	+, −	7
Concatenação	&	8
Relacionais	<, <=, >, >=, <>, =	9
Lógicos	Not	10
	And	11
	Or	12
	Xor	13
	Eqv	14
	Imp	15

Fonte: Produzido pelos autores.

Expressões também podem conter chamadas de função. Nesse caso, o token identificador da função é seguido por uma lista de argumentos entre parênteses. O algoritmo Shunting-yard adaptado trata o parêntese esquerdo como um marcador de função, empilhando o identificador e aguardando os argumentos. Se a função for do tipo `Function` (retorna valor) ou `Sub` (sem retorno), a compilação difere; `Sub`s são tratadas como comandos de chamada, e não podem ser chamados em expressões. Enquanto `Function`s produzem um valor, e portanto podem ser chamados tanto em expressões como em comandos de chamada.

Fora chamadas e operadores, expressões também podem ser compostas por valores literais.

Literais são tokens que representam valores constantes. O VB6 reconhece os seguintes tipos de literais:

- **Numéricos**: inteiros (ex.: `123`, `&H7B`) e de ponto flutuante (ex.: `3.14`, `1.5e-2`). Podem terminar com sufixos de tipo como `%` (Integer), `&` (Long), `!` (Single), `#` (Double), `@` (Currency).
- **Strings**: delimitadas por aspas duplas, ex.: `"Olá"`.
- **Booleanos**: `True` e `False`, que são simultaneamente palavras-chave e literais.

O compilador Ábaco converte cada literal para seu tipo nativo correspondente (inteiro de 32 bits, inteiro de 16 bits, `single`, `double`, `string` COM BSTR) e emite instruções para colocá-lo na pilha ou em registradores conforme a convenção `stdcall`.

3.2.6 Análise sintática de expressões

Expressões são analisadas usando um objeto denominado `ExprParser`, que recebe todo o código restante o armazena em seu estado interno. Esse objeto tem a capacidade de produzir o próximo token iterativamente, fazendo a análise do código sob demanda.

Em sua implementação `ExprParser` contém uma versão modificada do algoritmo Shunting-yard. Permitindo que receba código fonte textual em formato infixo, e resulte em uma sequência de tokens em formato pós-fixo, enquanto valida a entrada, retornado erros em caso de código inválido.

3.2.7 Comandos

Comandos são as unidades de código que não produzem valor e ocupam uma linha inteira. A análise sintática de uma linha consiste em identificar o comando principal baseando-se nos primeiros token da linha. Há duas categorias de comandos, declarativos e executáveis.

3.2.7.1 Comandos declarativos

Comandos declarativos não são traduzidos para código de máquina, mas fazem declarações que podem ser usadas por outros comandos, alguns exemplos de comandos declarativos são:

- **Rotinas**: `Function`, `Sub` `Property`.
- **Variáveis e constantes**: `Dim`, `Const`.
- **Tipos**: `Type`, `Enum`.
- **Opções de compilação**: `Option Explicit`, `Option Base 1`.

3.2.7.2 Comandos executáveis

Comandos executáveis são traduzidos para código de máquina que produz seus respectivos efeitos quando executadas, alguns exemplos de comandos executáveis são:

- **Atribuição**: `let` (implícito ou explícito). Ex.: `x = 10`.
- **Chamada de procedimento**: `Call MeuMétodo(argumentos)` ou simplesmente `MeuMétodo argumentos`.
- **Controle de fluxo**:
  - Condicional: `If ... Then ... Else ... End If`
  - Laços: `For ... Next`, `Do While ... Loop`, `Do Until ... Loop`, `While ... Wend`
  - Desvio: `Exit Do`, `Exit For`, `Exit Function`, `Exit Sub`, `End`
- **Outros**: `Debug.Print`

3.3 ARQUITETURA EM ETAPAS

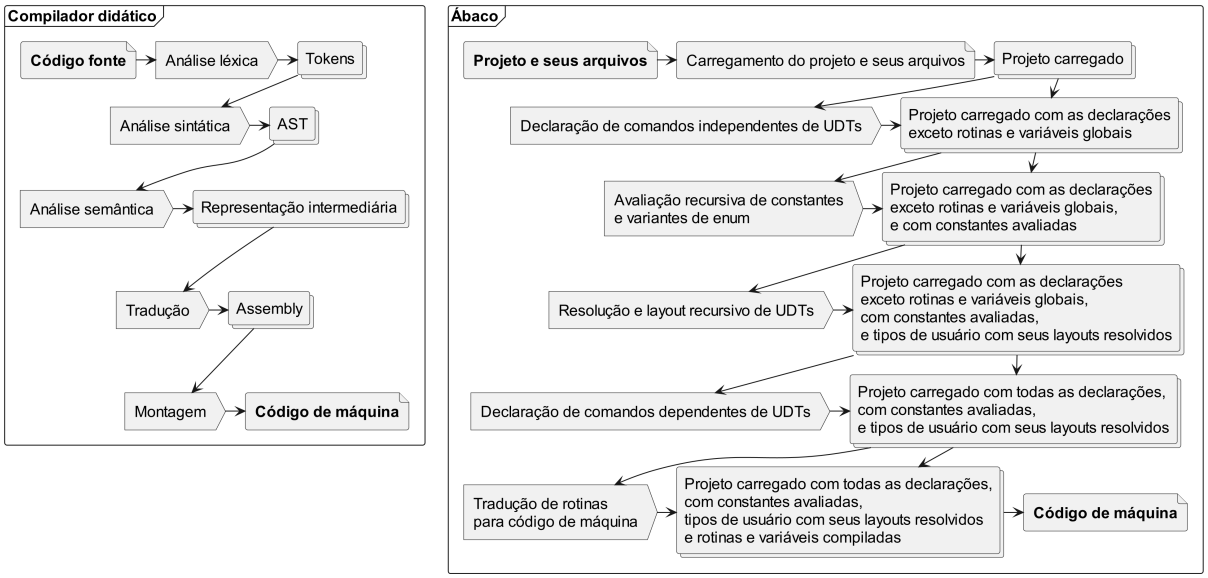
A implementação aqui apresentada é dividida em seis etapas, onde cada etapa precisa ser concluída antes que a próxima possa iniciar, e as etapas subsequentes usam as saídas das anteriores.

Compiladores acadêmicos ou didáticos costumam adotar uma divisão de etapas baseada em fases cronológicas rígidas, como o trabalho de Carvalho (2025), que usa cinco etapas, sendo elas: análise léxica, análise sintática, análise semântica, código intermediário e montagem.

Em contraste, optou-se por uma abordagem diferente. Visando a velocidade de compilação, a análise sob demanda foi escolhida, na qual o processamento do código do usuário é adiado até o último momento possível, exatamente onde o resultado da análise é necessário.

Fases cronológicas rígidas ainda aparecem no projeto, mas existem para garantir que as etapas subsequentes gozem de certas garantias de que precisam para funcionar, como está ilustrado na figura 1.

Figura 1 – Comparação de etapas de compilação



Fonte: Produzido pelos autores.

Um exemplo de uma etapa que precisa das garantias de etapas anteriores é a descrita na Seção 3.3.4, que precisa dos valores de constantes avaliados na etapa anterior, para algo como o tamanho de um *array* que afeta o layout de tipos definidos pelo usuário.

Essa abordagem reduz a necessidade de armazenar representações intermediárias na memória, reduzindo o gasto de memória, e por consequência o tempo necessário para a compilação.

Um exemplo disso é a análise sintática: no trabalho de Carvalho (2025), essa etapa é completamente concluída antes da análise semântica, visando o valor educacional. No entanto, quando implementada sob demanda, a consequência é que todas as seis etapas de compilação realizam alguma análise sintática, cada etapa fazendo apenas o mínimo para concluir suas demandas.

3.3.1 Carregamento do projeto e seus arquivos

O interpretador recebe na linha de comando o caminho do sistema operacional para um arquivo de projeto do VB6, com extensão "vbp", o arquivo é lido e todos os arquivos que são apontados por esse são carregados, concluindo essa etapa da compilação.

3.3.2 Declaração de comandos independentes de UDTs (Tipos definidos pelo usuário)

Nessa etapa, o código de todos os módulos passam por análise sintática, para identificar as porções de código que consistem as declarações que precisam ser extraídas nessa etapa.

As declarações são apenas identificadas, com seus nomes adicionados para listas internas, onde espaço foi alocado para cada uma, espaço este que será usado nas etapas seguintes.

Com isso feito já é possível discernir se *enums*, UDTs e constantes existem, o que é necessário na próxima etapa.

3.3.3 Avaliação recursiva de constantes e variantes de enum

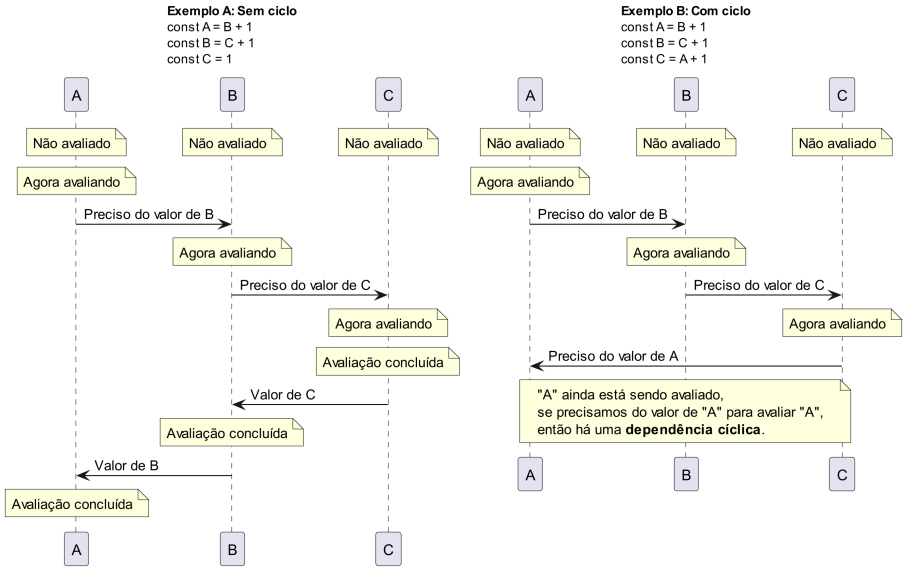
Com *enums* e constantes declarados, essa etapa usa a habilidade de verificar a existência e o espaço alocado para esses comandos para avaliar as expressões que assinalam valores a esses comandos.

Nessa etapa é necessário corretamente detectar e tratar dependências cíclicas, pois constantes podem depender de outras constantes de forma cíclica.

Para isso foi usado uma abordagem baseada no algoritmo branco, cinza, preto, que é capaz de detectar ciclos com apenas uma flag de três estados em cada nodo.

Todas as constantes começam como não avaliadas (branca), ao começarem a serem inicializadas, elas são marcadas como sendo avaliadas (cinza), e quando concluem são marcadas como avaliadas (preta). Enquanto são inicializadas (cinza), elas podem então usar o valor de outra constantes, estas podem estar não avaliadas, avaliando ou avaliadas. Caso estejam não avaliadas, a sua avaliação ocorre, recursivamente, caso já esteja sendo avaliada, um erro de dependência cíclica é emitido, pois a única forma em que isso pode ocorrer é se a constante depende diretamente ou indiretamente dela mesma, finalmente caso a constante já esteja avaliada, seu valor é usado. Um exemplo de sucesso e erro do procedimento está descrito na Figura 2.

Figura 2 – Exemplo de avaliação de constantes



Fonte: Produzido pelos autores.

3.3.4 Resolução e layout recursivo de UDTs

UDTs são declarações que definem um novo tipo, consistindo em uma estrutura de uma ou mais variáveis. Para o eventual uso desses tipos, é necessário calcular seu *layout*, que compreende a identificação dos campos especificados, suas posições relativas ao início da estrutura e o tamanho total ocupado pela estrutura.

Para que esses tipos sejam utilizados corretamente em chamadas à API do Windows ou em bibliotecas compartilhadas, é imprescindível que seu layout em memória seja compatível com o esperado por essas interfaces. O VB6 define que UDTs possuem layout sequencial, com alinhamento de membros conforme regras específicas. Conforme documentado em Alignment (2023), o compilador deve calcular o deslocamento de cada campo em relação ao início da estrutura, respeitando o alinhamento natural de cada tipo

A fórmula geral utilizada no Ábaco para este cálculo é a seguinte:

- O deslocamento de um campo é o menor múltiplo do seu alinhamento que seja maior ou igual ao deslocamento atual.
- O tamanho total da estrutura é o maior múltiplo do alinhamento máximo entre seus campos.

O Ábaco implementa esse cálculo recursivamente, pois um UDT pode conter outro UDT

como campo. A presença de referências cíclicas entre UDTs é tratada de forma análoga à avaliação de constantes, utilizando o algoritmo de marcação branco-cinza-preto para detectar e reportar dependências circulares. A figura 3 ilustra um exemplo concreto desse cálculo e da inserção de *padding*.

Figura 3 – Exemplo de layout de um UDT

Type Teste
A As **Integer**
B As **Byte**
C As **Long**
D As **Boolean**
End Type

Layout de Teste (12 Bytes)

Offset (Byte)	Dado	Posição no campo
+ 0	A (Integer)	Byte 1 de 2
+ 1	A (Integer)	Byte 2 de 2
+ 2	B (Byte)	Byte único
+ 3	*Padding*	
+ 4	C (Long)	Byte 1 de 4
+ 5	C (Long)	Byte 2 de 4
+ 6	C (Long)	Byte 3 de 4
+ 7	C (Long)	Byte 4 de 4
+ 8	D (Boolean)	Byte 1 de 2
+ 9	D (Boolean)	Byte 2 de 2
+ 10	*Padding*	
+ 11	*Padding*	

Cor	Variável	Tamanho
	A As Integer	2 Bytes
	B As Byte	1 Byte
	C As Long	4 Bytes
	D As Boolean	2 Bytes

Fonte: Produzido pelos autores.

3.3.5 Declaração de comandos dependentes de UDTs

Com os UDTs completamente descritos, comandos que dependem desses tipos podem ser declarados, estes consistem em rotinas e variáveis globais.

3.3.6 Tradução de rotinas para código de máquina

Nessa etapa, todas as rotinas são traduzidas para código de máquina por meio da tradução de cada linha de código. Cada comando na rotina é lido, e instruções nativas são emitidas que implementam a funcionalidade de cada um. Uma linha

de código pode precisar avaliar expressões, a compilação desta consiste em usar uma versão modificada do algoritmo Shunting-yard, proposto por Dijkstra (1961), para converter a notação infixa definida pela sequência de tokens em uma sequência pós-fixa, que pode então ser individualmente compilada para instruções nativas. Os operadores foram implementados como chamadas de função para código nativo externo ao código do usuário, visando simplificar a implementação. Todas as funções projetadas para essa finalidade foram nomeadas com o prefixo `callback`.

A tradução para o código de máquina foi projetado de tal forma que valores intermediários da avaliação de expressões sejam armazenados na pilha assim como uma implementação simples de avaliação de expressões usaria o algoritmo Shunting-yard. Essa abordagem simplifica a integração com este algoritmo e implementação como um todo. A simplificação se dá na possibilidade de projetar a tradução de cada valor, operador e comando de forma independente uns dos outros, exceto no que se refere à suas entradas e saídas.

A saída dessa etapa não é código *assembly*, mas sim código de máquina, optou-se por implementar um simples montador, capaz de montar as instruções que são emitidas nas etapas acima. Isso simplifica a execução JIT, pois elimina várias etapas custosas, como a concatenação de código *assembly*, a invocação de um montador externo, e o carregamento do artefato gerado.

Como o código de máquina se encontra em uma região de memória executável, para iniciar a execução do programa, basta obter o ponteiro para a primeira instrução escrita durante a compilação da função `main`, e chamar com a convenção de chamada `stdcall` e a assinatura da função `main`.

3.4 O MONTADOR

O montador foi implementado como uma estrutura capaz de alocar regiões executáveis e modificáveis de memória, ao emitir uma instrução, ela é montada em sua correta representação em bytes como define o Intel®... (2026), e é imediatamente escrita na sua posição final na memória.

A alocação foi implementada usando a API `VirtualAlloc` do Windows, que, entre outras capacidades, permite ao programa que chama adquirir regiões de memória com permissões de acesso customizadas, na implementação essa função foi chamada

para criar regiões de memória, legíveis, mutáveis e executáveis.

Também foi usada a API `FlushInstructionCache`, que deve ser chamada após escrever instruções nativas na memória, e antes de executá-las. Essa api inválida o cache de instruções da CPU, para garantir que a última versão da região de memória seja executada.

Como a instrução já sabe sua posição final, isso simplifica a emissão de instruções relativas como JMP, CALL, JCC, que dependem de onde são executadas na memória.

Para emitir instruções que fazem pulos para endereços absolutos ou endereços relativos antes dessa instrução, o deslocamento é calculado como o endereço de destino subtraído ao endereço após a instrução.

Em contraste, para emitir instruções que fazem pulos para endereços relativos após essa instrução, o endereço de onde o deslocamento é salvo em uma estrutura, para que quando a instrução alvo do pulo seja emitida, o deslocamento seja escrito usando a mesma subtração que acima.

3.5 AMBIENTE DE TESTE

Todos os experimentos de desempenho e validação foram conduzidos em um ambiente computacional controlado, cujas especificações são descritas a seguir, visando garantir a reprodutibilidade dos resultados.

- **Processador**: Intel Core i3-4005U (4ª geração), com frequência base de 1,70 GHz.
- **Memória RAM**: 4 GB DDR3.
- **Sistema operacional**: Microsoft Windows 11 IoT Enterprise LTSC (versão 10.0.26100, para a arquitetura x86 de 64 bits).
- **Linguagem e *toolchain***: Rust, versão 1.92.0, `i686-pc-windows-msvc`.
- **Implementação avaliada**: Interpretador JIT denominado "Ábaco", desenvolvido em Rust, compilado com otimizações.
- **Ambiente de referência**: Visual Basic 6 (VB6) interpretado via P-Code, utilizado para comparação nos benchmarks de execução.

O VB6 foi utilizado no modo interpretado P-Code (padrão para execução no ambiente de desenvolvimento), pois o Ábaco também não gera um executável independente. O modo de compilação nativa do VB6 não foi considerado, pois seu propósito é a distribuição final, não iteração rápida. Assim, a comparação com o P-Code é mais representativa do ciclo de desenvolvimento interativo, alvo deste trabalho.

3.5.1 Procedimento de aquecimento de cache

Para mitigar a influência de mecanismos de cache do sistema operacional e do hardware nos tempos de compilação e execução, adotou-se um procedimento de aquecimento prévio às medições efetivas. Esse procedimento consiste em executar o VB6 e Ábaco sobre os mesmos conjuntos de entrada um número determinado de vezes, descartando-se os tempos obtidos nessas iterações. A justificativa é que, durante as primeiras execuções, o sistema operacional carrega os arquivos do projeto para o cache de disco e a CPU ajusta suas estruturas de predição e cache de instruções. Ao aquecer o cache, as medições subsequentes refletem de forma mais fiel o desempenho nominal da implementação.

3.5.2 Medição da velocidade de compilação

Para avaliar a eficiência do processo de compilação just-in-time, foi utilizado um arquivo de código-fonte BASIC com 98.302 linhas, composto por 32.767 funções. Todas as funções são direta ou indiretamente chamadas pela função `Main`, garantindo que a totalidade do código seja efetivamente compilada durante a execução do programa.

O procedimento de medição seguiu as seguintes etapas:

1. Execução do Ábaco sobre o arquivo de teste por 50 vezes consecutivas, sem registro de tempos (fase de aquecimento de cache).
2. Após o aquecimento, nova bateria de 50 execuções, registrando-se o tempo de compilação de cada uma (desde o carregamento do projeto até a geração do código de máquina para todas as rotinas).
3. Cálculo da média aritmética dos tempos registrados.

Além do tempo total, foram calculadas duas métricas de vazão: linhas de código compiladas por segundo e megabytes processados por segundo.

3.5.3 Medição da velocidade de execução

A performance do código gerado pelo interpretador Ábaco foi comparada à execução nativa do VB6 (interpretado via P-Code) sobre quatro algoritmos clássicos, escolhidos por representarem diferentes padrões computacionais:

- **Crivo de Eratóstenes**: cálculo de números primos menores que 10.000.000, avaliando operações de laço e acesso a *arrays*.
- **Fibonacci recursivo**: implementação ingênua recursiva para o 35º termo, testando chamadas de função e sobrecarga de pilha.
- **Bubble sort**: ordenação de um *array* de 5.000 elementos, medindo comparações e trocas, o array é inicializado em ordem reversa, maximizando o trabalho do algoritmo.
- **FizzBuzz**: concatenação de 60.000 *strings* com condicionais.

O procedimento de medição foi:

1. Execução do Ábaco e do VB6 por 10 vezes (aquecimento de cache), descartando os tempos.
2. Nova bateria de 10 execuções para cada ambiente e cada algoritmo, registrando o tempo de execução (excluindo o tempo de compilação).
3. Cálculo da média aritmética dos tempos registrados.

3.5.4 Testes de validação funcional

Para verificar a corretude da implementação, foi elaborado um conjunto de testes funcionais que abrange as principais construções fundamentais da linguagem Visual Basic 6, com base nos princípios de Myers Tom Badgett (2012). Os testes foram organizados nas seguintes categorias, cada uma contendo múltiplos casos:

- Testes matemáticos básicos (operações aritméticas, precedência).
- Testes de lógica e comparação (operadores relacionais e lógicos).
- Testes de controle de fluxo (`If`, `For`, `Do While`, etc.).

- Testes de *strings* (concatenação).
- Testes de *arrays* (declaração, indexação, redimensionamento).
- Testes de chamadas de função simples (passagem por valor e por referência).
- Testes de conversões implícitas de tipos (inteiro para *string*, etc.).
- Testes de passagem `ByRef` (verificação de alteração de argumentos).
- Testes de `Type`, constantes e `Enum` (declaração e uso).
- Testes de chamada de API externa (por exemplo, `GetTickCount` da kernel32.dll).
- Testes de precedência e associatividade de operadores.
- Testes de determinismo da função `Rnd`.

Cada teste foi implementado como uma função VB6, compilada e executada tanto no VB6 original (para obter o comportamento esperado) quanto no Ábaco. Considerou-se "aprovado"quando o comportamento foi idêntico nos dois ambientes.

3.5.5 Estudo de caso integrado: Campo Minado

O livro de Myers Tom Badgett (2012) ressalta a importância de testar não apenas componentes individualmente, mas como eles interagem entre si. Visando isso, para validar o suporte a múltiplas funcionalidades simultaneamente, incluindo formulários, desenho GDI (Interface do dispositivo gráfico) e eventos do Windows, foi desenvolvido um programa completo em VB6: um jogo Campo Minado. Este programa foi escolhido por exigir a interação de diversas construções da linguagem e APIs do sistema, servindo como um teste de integração de alto nível.

O programa implementa as seguintes características:

- **Formulário principal**: O tabuleiro é desenhado manualmente via GDI.
- **Eventos de mouse**: captura cliques esquerdo (revelar célula) e direito (marcar bandeira) sobre o tabuleiro.
- **Lógica do jogo**: vetores bidimensionais, laços aninhados, constantes para configuração (tamanho, número de minas).

- **Redesenho dinâmico**: chamadas a funções da GDI (como `FillRect`, `BitBlt`, etc.) em resposta a eventos (como `Paint`, `MouseDown`, etc.).
- ***Buffering***: implementa o uso de um *bitmap*, onde o tabuleiro é desenhado antes de ser copiado para a tela.
- **Aleatoriedade**: as bombas são colocadas em locais aleatórios, mas o determinismo da função `Rnd` faz com que as bombas sempre apareçam no mesmo lugar. A função `Randomize` resolve isso, mas não foi chamada para ilustrar essa propriedade do `Rnd`.

Com a aplicação funcionando corretamente no VB6, ela foi executada também por meio do Ábaco.

4 RESULTADOS

Nesta seção, são apresentados os resultados obtidos nos experimentos descritos na metodologia, seguidos da discussão e das conclusões finais sobre a implementação do interpretador JIT Ábaco.

4.1 ANÁLISE DO CÓDIGO GERADO

Antes de detalhar as métricas de desempenho e a execução de testes complexos, é fundamental demonstrar a capacidade do interpretador Ábaco em traduzir o código-fonte para instruções nativas de forma correta. Para ilustrar o resultado prático desse processo de compilação, o Anexo B apresenta o *disassembly* completo de uma função de exemplo compilada pelo JIT.

A inspeção detalhada desse código de máquina permite observar como as estruturas de controle, a manipulação de memória e as operações de alto nível da linguagem original foram mapeadas para a arquitetura alvo.

4.2 RESULTADOS DE DESEMPENHO

4.2.1 Velocidade de compilação

A Tabela **??** resume as métricas de compilação obtidas após o aquecimento de cache.

Tabela 2 – Desempenho da compilação JIT (média de 50 execuções).

Métrica	Valor
Tempo médio de compilação	472 ms
Linhas compiladas	98.302
Tamanho do código fonte	2,86 MB
Vazão (linhas/segundo)	206.951
Vazão (MB/segundo)	6,05

Fonte: Produzido pelos autores.

Esses valores indicam que o interpretador Ábaco é capaz de processar uma quantidade substancial de código (quase 100 mil linhas) em menos de meio segundo, resultando em uma vazão superior a 200 mil linhas por segundo. Tal desempenho

é particularmente relevante para ciclos curtos de edição-execução (desenvolvimento iterativo).

4.2.2 Velocidade de execução

Os tempos médios de execução (em segundos) para cada algoritmo, nos ambientes VB6 e Ábaco, são apresentados na Tabela 3.

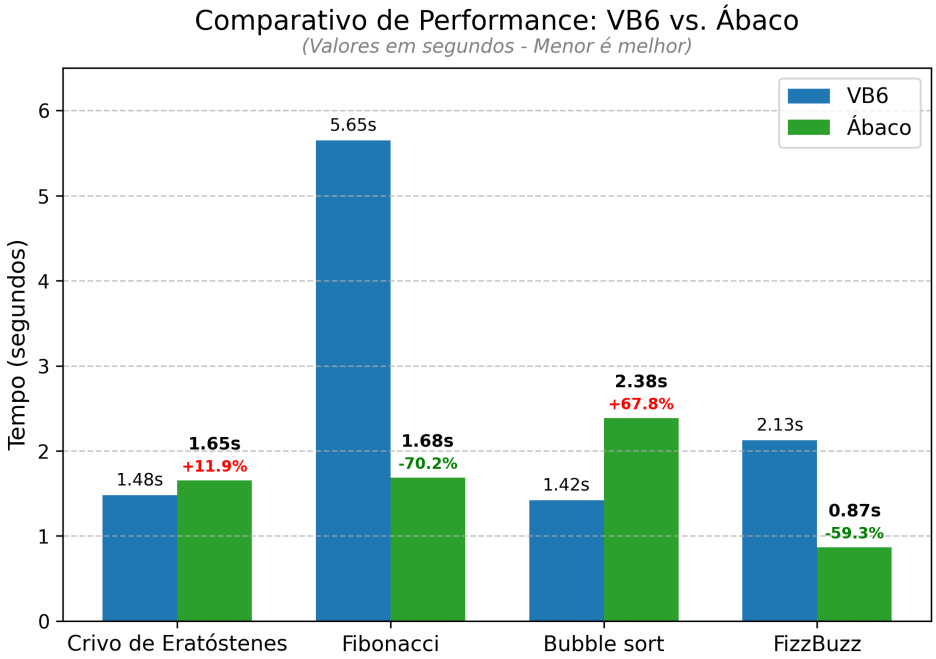
Tabela 3 – Comparação de tempos de execução (medidas em segundos).

Algoritmo	VB6 (P-Code)	Ábaco (JIT)
Crivo de Eratóstenes	1,477	1,653
Fibonacci recursivo	5,652	1,684
Bubble sort	1,421	2,384
FizzBuzz	2,127	0,866

Fonte: Produzido pelos autores.

Os resultados foram organizados em gráficos comparativos, conforme ilustrado na figura 4.

Figura 4 – Comparação de tempos de execução entre VB6 e Ábaco (médias em segundos).



Fonte: Produzido pelos autores.

Observa-se que o Ábaco supera significativamente o VB6 em dois algoritmos: Fibonacci recursivo (cerca de 3,4 vezes mais rápido) e FizzBuzz (aproximadamente 2,5

vezes mais rápido). No entanto, apresenta desempenho ligeiramente inferior no crivo de Eratóstenes (cerca de 12% mais lento) e no *bubble sort* (cerca de 68% mais lento).

4.2.3 Análise e interpretação dos resultados

A diferença mais expressiva a favor do Ábaco ocorre no Fibonacci recursivo. Uma hipótese para explicar essa vantagem é que o VB6, ao interpretar P-Code, insere verificações e tratamento de exceções a cada chamada de função, mesmo quando nenhuma exceção é lançada. O Ábaco, em sua implementação atual, não implementa tratamento de exceções, sendo uma possível causa para diferença de performance.

No caso do *bubble sort* e do crivo de Eratóstenes, onde o VB6 apresenta melhor desempenho, suspeita-se que o acesso a *arrays* e a manipulação de índices sejam mais otimizados no ambiente nativo do VB6. O Ábaco ainda utiliza uma estratégia de acesso a *arrays* com verificação de limites genérica, que pode não ser tão eficiente quanto a do VB6.

No FizzBuzz, a vantagem do Ábaco pode ser atribuída à concatenação de *strings* e à geração de código de máquina direto, enquanto o VB6 interpreta operações de *string* via P-Code, que é intrinsecamente mais lento.

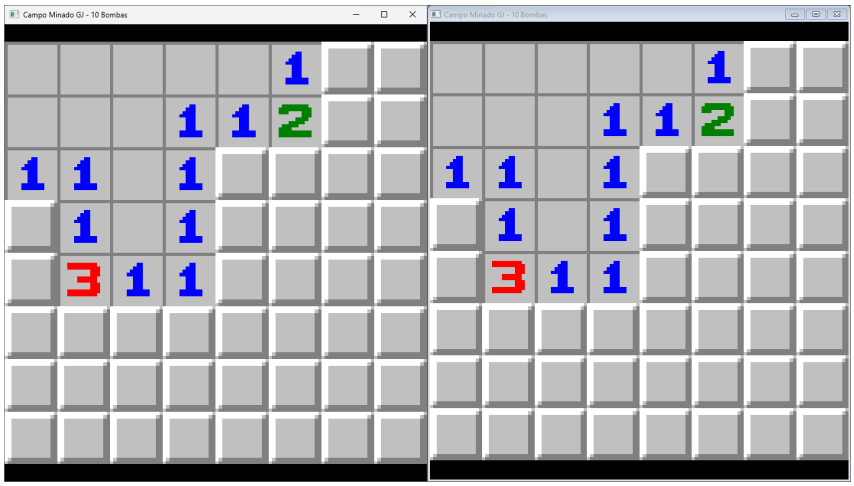
4.2.4 Resultados dos testes funcionais

Todos os 12 conjuntos de testes funcionais foram executados com sucesso no interpretador Ábaco. Cada caso de teste produziu exatamente a mesma saída (valores, mensagens ou efeitos) que o ambiente VB6 original. Isso demonstra que, dentro do subconjunto de funcionalidades implementado, o interpretador está semanticamente correto e compatível com a especificação do VB6.

4.3 EXECUÇÃO DO TESTE DE INTEGRAÇÃO - CAMPO MINADO

O programa Campo Minado descrito na metodologia (Seção 3.5.5) foi executado no interpretador Ábaco. A Figura 5 mostra o jogo em execução.

Figura 5 – Campo Minado executando no Ábaco à esquerda, e no VB6 à direita.



Fonte: Produzido pelos autores.

Na comparação com a execução no VB6 (Figura 5), o comportamento visual e a jogabilidade são idênticos. Todas as interações testadas (revelar células, marcar bandeiras, reiniciar) produziram os mesmos resultados.

O programa sendo executado no Ábaco demonstra que:

- A compilação JIT tem suporte básico à formulários.
- As chamadas às APIs GDI (`CreateSolidBrush`, `SelectObject`, `FillRect`, `BitBlt`) são resolvidas corretamente.
- O sistema de eventos (tratamento de mensagens `WM_LBUTTONDOWN`, `WM_RBUTTONDOWN`) está integrado ao laço de mensagens.
- A ABI (*Application Binary Interface*) de UDTs foi corretamente implementada pois às APIs GDI conseguem utilizar os UDTs.
- As capacidades básicas da linguagem funcionam, como constantes, *enums*, declarações externas, funções, etc.

A execução bem-sucedida desse programa fornece evidência qualitativa de que a implementação alcançou um nível de maturidade suficiente para aplicações interativas não triviais.

4.3.1 Desempenho observado

O tempo de compilação do formulário principal (cerca de 1000 linhas de código) foi de 10 ms, e a resposta a eventos de clique manteve-se fluida, sem atrasos percep-

tíveis. O uso de redesenho GDI a cada clique não apresentou *tearing* nem atrasos visuais, indicando que o acoplamento entre o código compilado e as APIs Windows é eficiente.

Esse resultado confirma que a implementação é robusta para uma aplicação gráfica interativa de média complexidade.

5 CONCLUSÃO

Este trabalho apresentou a implementação de um interpretador JIT para a linguagem Visual Basic 6, denominado Ábaco, utilizando a linguagem Rust e direcionado à arquitetura x86 de 32 bits no sistema operacional Windows. A arquitetura adotada baseou-se em análise sob demanda, que permitiu reduzir o consumo de memória e o tempo de compilação ao evitar a construção de representações intermediárias completas.

Os resultados experimentais mostraram que:

- O Ábaco é capaz de compilar cerca de 200 mil linhas de código por segundo, com tempo médio de 472 ms para um projeto de 98 mil linhas, o que o torna adequado para uso interativo.
- A velocidade de execução do código gerado pelo Ábaco é comparável com a velocidade de interpretação do VB6, dependendo de que comandos são executados, mas com muitas oportunidades de otimização futura.
- A corretude funcional foi verificada por meio de uma suíte de testes, na qual o Ábaco obteve 100% de aprovação.

Em suma, o interpretador Ábaco demonstra que é viável obter ganhos de desempenho substanciais em relação ao interpretador VB6 original, mantendo compatibilidade semântica e tempos de compilação reduzidos. A abordagem de compilação JIT *eager* com análise sob demanda mostrou-se eficaz para essa linguagem de natureza predominantemente linear (baseada em linhas). Este trabalho pode servir de base tanto para ferramentas de modernização de código VB6 quanto para estudos de compilação de linguagens legadas.

6 TRABALHOS FUTUROS

A implementação atual do compilador Ábaco, embora funcional para um subconjunto significativo da linguagem VB6, ainda apresenta diversas limitações. Estas limitações definem naturalmente as direções para a continuidade do desenvolvimento, sendo organizadas nos seguintes itens pendentes:

1. **Tratamento de exceções**: o suporte às construções `On Error GoTo`, `Resume` e `Err` não foi implementado, o que impede a execução de código que dependa de recuperação de erros.

2. **Depuração**: não há suporte a breakpoints, inspeção de variáveis, execução passo a passo ou *hot-reloading*, recursos essenciais para o desenvolvimento iterativo.

3. **Ampliação do conjunto de construções da linguagem**: diversos comandos (`Open`, `Input`, `Line Input`, `Print`), funções da biblioteca padrão (manipulação de arquivos, datas, formatação) e controles ActiveX ainda não são suportados.

4. **Extensões além do VB6 original**: a linguagem poderia ser melhorada com suporte a *multithreading*, tipos genéricos ou interoperabilidade com bibliotecas modernas - funcionalidades inexistentes no VB6 e que justificam uma ferramenta livre.

5. **Otimização do código gerado**: a qualidade das instruções nativas emitidas é ainda básica; técnicas como propagação de constantes e alocação de registradores podem melhorar significativamente a performance.

6. **Portabilidade para x86-64**: a versão atual é restrita à arquitetura x86 de 32 bits; a migração para 64 bits permitiria acesso a mais memória e melhor desempenho em sistemas modernos.

O endereçamento desses itens, seja em sua totalidade ou em parte, constitui o caminho natural para que o Ábaco se torne uma alternativa viável e superior ao ambiente VB6 original, combinando a compatibilidade com código legado e as vantagens de uma implementação moderna e aberta.

REFERÊNCIAS

ALA-HULKKO, T. Modernization of a legacy codebase. 2025. Disponível em: <https://www.theseus.fi/handle/10024/894263>. Acesso em: 2 jun. 2026. Citado na página 15.

ALFRED, V. *et al.* **Compilers Principles, Techniques**. Pearson, 2007. Disponível em: <http://www.ir.harambeeuniversity.edu.et/bitstream/handle/123456789/1534/Compilers%20Principles%2C%20Techniques%20%26%20Tools%20by%20Alfred%20V.%20Aho.%20%28%20PDFDrive.com%20%29.pdf?sequence=1&isAllowed=y>. Acesso em: 2 jun. 2026. Citado na página 18.

ALIGNMENT. 2023. Disponível em: <https://learn.microsoft.com/en-us/cpp/cpp/alignment-cpp-declarations?view=msvc-170>. Acesso em: 29 mai. 2026. Citado na página 31.

CALLING Conventions. 2021. Disponível em: <https://learn.microsoft.com/en-us/cpp/cpp/calling-conventions?view=msvc-170>. Acesso em: 29 mai. 2026. Citado na página 21.

CARVALHO, P. H. F. Micro c-um compilador acadêmico. Fundação Universidade Federal de Mato Grosso do Sul, 2025. Acesso em: 29 mai. 2026. Citado 2 vezes nas páginas 28 e 29.

CHUBCHENKO, S. **Vb Decompiler**. 2025. Disponível em: <https://www.vb-decompiler.org/vb60_reversing.htm>. Acesso em: 29 mai. 2026. Citado na página 17.

CORMEN, T. H. *et al.* **Introduction to algorithms**. MIT press, 2022. Disponível em: <https://books.google.com.br/books?id=drZNEAAAQBAJ>. Acesso em: 2 jun. 2026. Citado na página 20.

DALLE, J.-M.; JULLIEN, N. Open-source vs. proprietary software. **Guest lecture at ESSID Summer School, Cargèse**, 2001. Disponível em: <https://www.researchgate.net/publication/2559454_Open-Source_vs_Proprietary_Software>. Acesso em: 2 jun. 2026. Citado na página 15.

DEVI, S. L.; PERUMAL, R. S.; MOULI, P. C. Challenges and issues in code migration from vb 6.0 to vb .net. 2011. Disponível em: <https://www.researchgate.net/publication/235899034_Challenges_and_Issues_in_Code_Migration_From_VB_60_TO_VBNET>. Acesso em: 2 jun. 2026. Citado na página 15.

DIJKSTRA, E. W. **Algol 60 translation : An Algol 60 translator for the x1 and Making a translator for Algol 60**. [S.l.], 1961. Disponível em: <http://www.cs.utexas.edu/users/EWD/MCReps/MR35.PDF>. Acesso em: 2 jun. 2026. Citado 2 vezes nas páginas 20 e 33.

FREEBASIC. 2026. Disponível em: <https://www.freebasic.net/>. Acesso em: 29 mai. 2026. Citado na página 14.

INTEL® 64 and IA-32 Architectures Software Developer's Manual. 2026. Disponível em: <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>. Acesso em: 29 mai. 2026. Citado 2 vezes nas páginas 21 e 33.

KOTZMANN, T. *et al.* Design of the java hotspot™ client compiler for java 6. **ACM Transactions on Architecture and Code Optimization (TACO)**, ACM New York, NY, USA, v. 5, n. 1, p. 1–32, 2008. Disponível em: <https://dl.acm.org/doi/abs/10.1145/1369396.1370017>. Acesso em: 2 jun. 2026. Citado na página 19.

LUAJIT. 2025. Disponível em: <https://luajit.org/>. Acesso em: 2 jun. 2026. Citado na página 19.

MS-VBAL: VBA Language Specification. 2025. Disponível em: <https://learn.microsoft.com/en-us/openspecs/microsoft_general_purpose_programming_languages/ms-vbal/d5418146-0bd2-45eb-9c7a-fd9502722c74>. Acesso em: 29 mai. 2026. Citado na página 15.

MYERS TOM BADGETT, C. S. G. J. The art of software testing. In: ____. John Wiley & Sons, Ltd, 2012. ISBN 9781119202486. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119202486>. Acesso em: 2 jun. 2026. Citado 2 vezes nas páginas 36 e 37.

RADBASIC. 2026. Disponível em: <https://www.radbasic.dev/>. Acesso em: 29 mai. 2026. Citado na página 15.

SPIDERMONKEY Javascript Engine. 2026. Disponível em: <https://spidermonkey.dev/>. Acesso em: 2 jun. 2026. Citado na página 19.

TWINBASIC. 2026. Disponível em: <https://twinbasic.com/>. Acesso em: 29 mai. 2026. Citado na página 15.

V8 Javascript Engine. 2026. Disponível em: <https://v8.dev/>. Acesso em: 2 jun. 2026. Citado na página 19.

ANEXO A – DEFINIÇÃO DA SINTAXE TODAS AS LINHAS SUPORTADAS PELO ÁBACO

```
Lista de todas as linhas suportadas pela implementação atual da análise sintática de
 ↪ linhas, omitindo as palavras chave Private, Public e Friend onde aplicável.

Legenda:
"[N]" indica um nome a ser definido pelo usuário,
"[P]" indica uma lista de parâmetros,
"[E]" indica uma expressão,
"[T]" indica um tipo,
"[E],..." indica uma lista de expressões separadas por vírgula.

Option Explicit
Option Compare Binary
Option Compare Text
Option Base 0
Option Base 1
Option Private Module
Declare Function Lib "[N]" Alias "[N]" [N]([P]) As [T]
Declare Sub Lib "[N]" Alias "[N]" [N]([P])
Type [N]
End Type
Enum [N]
End Enum
Function [N]([P]) As [T]
Sub [N]([P])
Property Get [N]([P])
Property Let [N]([P])
Debug.Print [E]
End Function
End Sub
End Property
Dim [N] As [T]
Const [N] As [T]
Let [E] = [E]
[E] = [E] 'atribuição sem Let
Call [E]([E],...)
[E] [E],... 'chamada sem Call nem parêntesis
If [E] Then
ElseIf [E] Then
Else
End If
Do
Do While [E]
```

```
Do Until [E]
Loop
Loop While [E]
Loop Until [E]
While [E]
Wend [E]
For [N] = [E] To [E] Step [E]
Next
Next [E]
Exit Sub
Exit Function
Exit Do
Exit For
ReDim [E]([E],...) As [T]
ReDim Preserve [E]([E],...) As [T]
Erase [E]
```

ANEXO B – EXEMPLO DO CÓDIGO DE MÁQUINA GERADO PARA UMA ROTINA DE EXEMPLO

```
Sub Main()
Dim SimOuNao As Boolean
Dim Moeda As String
Randomize
SimOuNao = Rnd < 0.5!
If SimOuNao Then
    Moeda = "Cara"
Else
    Moeda = "Coroa"
End If
Debug.Print "A moeda deu " & Moeda
End Sub

; Sub Main()
55             | push ebp    ; ┐
8BEC           | mov ebp,esp ; │
53             | push ebx    ; ├ (Prólogo da função)
56             | push esi    ; │
57             | push edi    ; ┘
31C0           | xor eax,eax ; ┐
50             | push eax    ; ├ (Alocando espaço para as variáveis)
50             | push eax    ; ┘
; Randomize
E8 B9EA35FF    | call callback_randomize
; SimOuNao = Rnd < 0.5!
68 0000003F    | push 3F000000 ; (0.5!)
E8 A9FC8DFF    | call callback_rnd       ; ┐
81C4 FCFFFFFF  | sub esp, 4              ; ├ (Rnd)
D91C24         | fstp dword ptr ss:[esp] ; ┘
E8 2B638FFF    | call callback_cmp_single_to_single ; ┐
C1F8 1F        | sar eax,1F                         ; ├ (<)
50             | push eax                           ; ┘
58             | pop eax     ; ┐
F7D8           | neg eax     ; ├ conversão para booleano
19C0           | sbb eax,eax ; │
50             | push eax    ; ┘
58             | pop eax                      ; ┐ (SimOuNao =)
66:8945 F2     | mov word ptr ss:[ebp-E], ax ; ┘
; If SimOuNao Then
0FBF45 F2      | movsx eax,word ptr ss:[ebp-E] ; ┐ (SimOuNao)
50             | push eax                     ; ┘
58             | pop eax         ; ┐
85C0           | test eax,eax    ; ├ (If ... Then)
```

```
75 05         | jne then_branch ;
E9 20000000   | jmp else_branch ;
; S = "Cara"
then_branch:
68 C4934001   | push 14093C4             ; (*14093C4 = "Cara")
E8 C3EA35FF   | call callback_clone_bstr ; ("Cara")
50            | push eax                ;
FF75 EC       | push dword ptr ss:[ebp-14] ; (Moeda = ...)
E8 4AEA35FF   | call callback_free_bstr    ; (Liberação da memória de Moeda)
8F45 EC       | pop dword ptr ss:[ebp-14]  ; (Moeda = ...)
; Else
E9 1B000000   | jmp end_if ; (Else)
; Moeda = "Coroa"
else_branch:
68 EC934001   | push 14093EC             ; (*14093EC = "Coroa")
E8 A3EA35FF   | call callback_clone_bstr ; ("Coroa")
50            | push eax                ;
FF75 EC       | push dword ptr ss:[ebp-14] ; (Moeda = ...)
E8 2AEA35FF   | call callback_free_bstr    ; (Liberação da memória de Moeda)
8F45 EC       | pop dword ptr ss:[ebp-14]  ; (Moeda = ...)
end_if:
; Debug.Print "A moeda deu " & Moeda
68 FC844001   | push 14084FC             ; (*14084FC = "A moeda deu ")
E8 88EA35FF   | call callback_clone_bstr ; ("A moeda deu ")
50            | push eax                ;
FF75 EC       | push dword ptr ss:[ebp-14] ;
E8 7FEA35FF   | call callback_clone_bstr   ; (Moeda)
50            | push eax                  ;
E8 096636FF   | call callback_cat_bstr_bstr_reversed ; (&) (Operador de
 ↪ concatenação)
E8 E3CF32FF   | call callback_debug_print ; (Debug.Print)
; End Sub
FF75 EC       | push dword ptr ss:[ebp-14] ; (Moeda = ...)
E8 F6E935FF   | call callback_free_bstr    ; (Liberação da memória de Moeda)
81C4 08000000 | add esp, 8 ; (Liberando espaço das variáveis)
5F            | pop edi      ;
5E            | pop esi      ;
5B            | pop ebx      ; (Epílogo da função)
8BE5          | mov esp, ebp ;
5D            | pop ebp      ;
C3            | ret          ;
```