Trabalho Prático – Compilador para a Linguagem SL

Relatório da Etapa 2: Análise Semântica e Interpretador

Luiz Eduardo Fugliaro - 22.1.4014 — Gustavo Zacarias de Souza - xx.x.xxxx

20 de Fevereiro de 2026

1 Introdução

Este relatório descreve o desenvolvimento da Etapa 2 do trabalho prático da disciplina de Compiladores, cujo objetivo foi a implementação da análise semântica e do interpretador para a linguagem SL.

Conforme especificado no enunciado, esta etapa envolveu:

- Projeto e implementação do sistema de tipos;
- Implementação do analisador semântico;
- Implementação do interpretador para execução de programas SL;
- Verificação das regras semânticas obrigatórias da linguagem.

Este relatório apresenta o que foi implementado, o que não foi implementado e como ferramentas baseadas em modelos de linguagem foram utilizadas durante o desenvolvimento.

2 O que foi feito

2.1 Sistema de Tipos

Foi projetado e implementado um sistema de tipos estático para a linguagem SL, contemplando:

- Tipos primitivos: `int`, `float`, `bool`, `string`;
- Arranjos unidimensionais;
- Registros (structs);
- Funções com parâmetros e tipo de retorno;

- Polimorfismo paramétrico (generics);
- Inferência de tipos em declarações com tipo omitido.

A verificação de tipos ocorre antes da execução, garantindo que apenas programas semanticamente válidos sejam interpretados.

2.2 Analisador Semântico

Foi implementado um analisador semântico responsável por:

- Controle de escopos léxicos por meio de pilha de ambientes;
- Verificação de declaração única de identificadores no mesmo escopo;
- Verificação de compatibilidade de tipos em operações aritméticas, relacionais e booleanas;
- Verificação de número e tipos de argumentos em chamadas de função;
- Verificação de acesso válido a campos de registros;
- Verificação de acesso válido a elementos de arranjos;
- Verificação de compatibilidade entre tipo de retorno e tipo declarado da função;
- Coleta prévia de declarações para permitir chamadas antecipadas de funções.

A implementação garante que erros semânticos sejam detectados antes da execução do programa.

2.3 Suporte a Generics e Inferência de Tipos

Foi implementado suporte a polimorfismo paramétrico, permitindo a definição de funções genéricas com inferência automática dos tipos concretos a partir dos argumentos fornecidos.

Além disso, foi implementado mecanismo de inferência de tipos para declarações onde o tipo é omitido, desde que haja expressão inicializadora.

2.4 Interpretador

Foi desenvolvido um interpretador para execução direta da AST gerada pelo parser.

O interpretador implementa:

- Execução de funções com escopo próprio;

- Execução de estruturas de controle (`if`, `while`);
- Manipulação de arranjos com verificação de limites;
- Manipulação de registros em tempo de execução;
- Operações aritméticas, relacionais, booleanas e concatenação de strings;
- Comando `print`;
- Propagação correta de comandos `return`.

A execução ocorre em um ambiente com pilha de escopos, garantindo isolamento adequado de variáveis locais.

3 O que não foi feito

Embora a maior parte dos requisitos da Etapa 2 tenha sido implementada, alguns pontos não foram completamente desenvolvidos:

- O interpretador não possui implementação completa da estrutura `for`;
- Atribuições complexas envolvendo campos internos de structs não foram totalmente suportadas;
- Inicialização literal de arranjos (por exemplo, `[1,2,3]`) não foi implementada no interpretador;
- O tratamento de erros em tempo de execução utiliza abortos diretos da execução em vez de um mecanismo estruturado de exceções.

Apesar dessas limitações, todos os requisitos obrigatórios explicitamente descritos para a Etapa 2 foram atendidos.

4 Uso de Prompts e Ferramentas de IA

Durante o desenvolvimento da Etapa 2, foram utilizadas ferramentas baseadas em Modelos de Linguagem (LLMs) tanto para esclarecimento de dúvidas conceituais quanto para apoio na implementação de trechos do compilador. Conforme solicitado na especificação do trabalho, esta seção detalha os prompts realizados, os resultados obtidos e como estes foram utilizados no desenvolvimento do projeto.

As consultas realizadas podem ser divididas em dois grupos:

- Consultas para esclarecimento de dúvidas específicas (modelagem, arquitetura e organização do código);
- Consultas para produção ou esboço inicial de implementações.

Todo código gerado foi revisado, adaptado e integrado manualmente ao projeto.

4.1 1. Planejamento Inicial da Etapa 2

Prompt 1: Considere agora que irei prosseguir com o trabalho de implementação do compilador para a linguagem SL, após ter implementado a análise léxica e sintática. Agora, para a Etapa 2, o seguinte arquivo descreve as especificações de implementação do próximo trabalho. Analise-o e me guie para dar início.

Resultado: Análise da especificação e orientação inicial sobre como estruturar o desenvolvimento.

Utilização: Serviu como ponto de partida estratégico para organizar as tarefas da Etapa 2.

Prompt 2: Me forneça um resumo dos pontos a serem implementados e uma breve explicação deles, seguindo uma ordem de implementação lógica.

Resultado: Roteiro prático das necessidades de implementação, com breve explicação de cada passo seguindo uma ordem lógica.

Utilização: Utilizado como checklist estruturado para conduzir o desenvolvimento.

4.2 2. Implementação do TypeChecker

Prompt 3: Me forneça a estrutura do arquivo TypeChecker com as descrições dos métodos necessários, de maneira que me guie na implementação.

Resultado: Estrutura inicial do arquivo de verificação de tipos, clareando o entendimento e agilizando o desenvolvimento.

Utilização: Base estrutural para criação do módulo TypeChecker.hs.

Prompt 4: Considerando a tabela de símbolos, a mônada de verificação, as funções auxiliares que controlam os escopos e o verificador de expressões checkExpr, como eu poderia implementar a função checkStmt para verificar comandos? Faça uma explicação da lógica apresentada.

Resultado: Explicação da lógica de verificação de comandos e proposta de implementação do método checkStmt.

Utilização: Serviu como base conceitual para implementação da verificação de comandos.

Prompt 5: Agora com checkStmt e checkExpr, o que falta para que o TypeChecker comece a checar tipos?

Resultado: Indicação da necessidade de um ponto de entrada que percorra as declarações de topo (método checkProgram).

Prompt 6: Como ficaria a implementação desse método checkProgram e como eu faria para vinculá-lo à Main do meu programa?

Resultado: Implementação sugerida do checkProgram e orientação para integração com a Main.

Prompt 7: Através deste arquivo Main.hs que eu já possuo, inclua as alterações necessárias para que o código consiga receber o comando para o verificador de tipos, mantendo a estrutura já presente.

Resultado: Arquivo Main.hs adaptado para incluir a execução do verificador de tipos.

Utilização: Integração do TypeChecker ao pipeline do compilador.

4.3 3. Construção e Validação com Testes

Prompt 8: Baseado na especificação da linguagem SL, me forneça arquivos para que eu consiga testar a minha implementação. Também indique quais cenários o verificador deve reconhecer.

Resultado: Arquivos de teste:

- teste_sucesso_completo.sl
- teste_erro_tipos.sl
- teste_erro_funcao.sl
- teste_erro_escopo.sl
- teste_erro_condicao.sl

Prompt 9: Me forneça um arquivo que contenha um erro com struct e array.

Resultado: teste_erro_struct_array.sl

Prompt 10: Agora que todos os casos de teste passaram pelo compilador, baseado no que eu já implementei, verifique se está atendendo às necessidades da Etapa 2.

Resultado: Confirmação de atendimento às especificações do TypeChecker e indicação do próximo passo: implementação do interpretador.

4.4 4. Implementação do Interpretador

Prompt 11: Irei enviar o meu arquivo AST.hs e preciso que me forneça as necessidades de implementação do arquivo Interpreter.hs.

Resultado: Definição dos conceitos necessários:

- Representação de valores em memória;
- Ambiente de execução;
- Função de avaliação como máquina de execução.

Prompt 12: Crie uma estrutura básica desses três componentes necessários para executar uma conta simples.

Resultado: Interpreter.hs capaz de executar contas básicas (ex.: 3 + 5).

Prompt 13: Me forneça um arquivo teste com contas simples.

Resultado: teste_execucao.sl

Prompt 14: Ainda é necessário tratar arrays e structs. Quais mudanças seriam necessárias em evalExpr e evalStmt?

Resultado: Indicação de atualização em evalExpr para leitura e criação de structs e adaptação de evalStmt para atribuições complexas.

Prompt 15: Adicione essas modificações nas minhas funções e explique o novo funcionamento.

Resultado: evalExpr e evalStmt atualizados.

Prompt 16: Me forneça um arquivo de teste que apresente estes novos casos.

Resultado: teste_complexo.sl

Prompt 17: Me forneça outros exemplos para testar completamente o interpretador.

Resultado:

- bubble_sort.sl
- fibonacci.sl
- scope_test.sl

4.5 5. Implementação de Generics e Inferência

Prompt 18: O que deve ser adicionado ao código para implementar generics e inferência de tipos?

Resultado: Necessidade de:

- Alterar a AST para suportar tipos genéricos e tipos inferidos;
- Implementar substituição de tipos genéricos no TypeChecker;
- Implementar lógica de inferência para tipos implícitos.

Prompt 19: Adicionei TyGeneric String e TyAuto na AST. Está correto? Falta algo?
Resultado: Indicação da necessidade de alterar FuncDecl para armazenar a lista de parâmetros genéricos.

Prompt 20: O que deve ser adicionado ao Parser?
Resultado: Leitura de:

- Parâmetros genéricos (¡T¿);
- Declarações com inferência (let x = ...);
- Uso de tipos genéricos no código.

Prompt 21: Aplique essas implementações no meu código, explicando cada parte.
Resultado: Parser atualizado com as novas funcionalidades.

Prompt 22: O cabal build rodou corretamente. O que posso fazer agora?
Resultado: Executar teste de inferência (teste_inferencia.sl).

Prompt 23: Como implementar a substituição de generics?
Resultado: Criação de função auxiliar instantiate para substituir tipos genéricos por tipos concretos durante a chamada da função.

Prompt 24: Realize essas implementações no TypeChecker e explique o funcionamento.

Prompt 25: Me forneça arquivos de teste para validar generics.
Resultado:

- teste_generics_real.sl
- erro_generics.sl

Prompt 26: O erro_generics falhou como esperado e o teste_generics_real rodou corretamente.
Resultado: Conclusão de que o Front-end (Análise Léxica e Sintática) e o Middle-end (Análise Semântica e Interpretação) estavam concluídos conforme as especificações da Etapa 2.

5 Divisão de Tarefas

A divisão de tarefas foi realizada da seguinte forma:

- Gustavo: Projeto e implementação do sistema de tipos e analisador semântico;
- Luiz: Implementação do interpretador e testes;
- Ambos: Integração, testes finais e elaboração do relatório.

6 Testes Automatizados do Verificador de Tipos

Esta seção detalha a suíte de testes implementada para validar as regras semânticas da linguagem SL. O foco principal é garantir que o analisador semântico identifique corretamente erros de tipagem e escopo, além de validar construções sintáticas complexas.

6.1 Resumo dos Casos de Teste

A tabela abaixo descreve os arquivos de teste utilizados e o comportamento esperado de cada um no processo de compilação.

Arquivo	**Objetivo**	**Descrição Técnica**
`test_sucesso_completo.sl`	Sucesso	Valida structs, recursão, arrays e expressões lógicas.
`test_erro_funcao.sl`	Erro	Captura retornos inválidos e erro no número/tipo de argumentos.
`test_erro_escopo.sl`	Erro	Garante que variáveis declaradas dentro de blocos (`if`) não vazem para o escopo externo.
`test_erro_tipos.sl`	Erro	Proíbe operações aritméticas entre tipos incompatíveis (`int` + `bool`).
`test_erro_condicao.sl`	Erro	Exige estritamente tipo `bool` em estruturas de controle (`if`/`while`).

Table 1: Resumo da bateria de testes semânticos.

6.2 Automação com Makefile

Para otimizar o fluxo de desenvolvimento, foi criado um `Makefile` que automatiza a execução do compilador sobre a suíte de testes. O comando principal utiliza o gerenciador de pacotes `cabal` para rodar o executável `sl-compiler` com a flag `--check`, que foca exclusivamente na análise semântica.

Listing 1: Conteúdo do Makefile para automação de testes

```
TESTS = \
    src/tests/test_sucesso_completo.sl \
    src/tests/test_erro_tipos.sl \
    src/tests/test_erro_funcao.sl \
    src/tests/test_erro_escopo.sl \
    src/tests/test_erro_condicao.sl
```

6.3 Execução

Para rodar todos os testes e verificar a integridade do verificador de tipos, execute no terminal:

```
make test
```

7 Conclusão

A Etapa 2 foi concluída com sucesso, contemplando a implementação do sistema de tipos, do analisador semântico e de um interpretador funcional para a linguagem SL.

O compilador nesta etapa é capaz de detectar erros semânticos antes da execução e interpretar corretamente programas válidos contendo funções, estruturas de controle, arranjos, registros e generics.

Os objetivos definidos no enunciado para esta etapa foram atendidos.