RELATÓRIO DO TRABALHO COMPILADOR E INTERPRETADOR PARA A LINGUAGEM LANG

BCC328 - Construção de compiladores I

Autores:
19.2.4004 - Mateus Henrique Máximo Lima de Souza
19.2.4008 - Luiz Fernando Rodrigues Fernandes
24.1

1 Introdução

Este trabalho apresenta o desenvolvimento de um analisador léxico, analisador sintático, analisador semântico e de um interpretador para a linguagem Lang. O objetivo principal deste relatório é dar um embasamento teórico sobre cada fase e especificar o processo de criação desses componentes para a linguagem Lang.

1.1 Análise Léxica

A análise léxica é a primeira fase do processo de compilação, onde o código fonte é transformado em uma sequência de tokens. Cada token representa uma unidade atômica do código, como palavras-chave, identificadores, operadores e literais. A função principal do analisador léxico (lexer) é ler o código fonte, identificar os padrões léxicos e categorizá-los corretamente.

Nesta etapa, o lexer percorre o código fonte caractere por caractere, agrupando-os em tokens conforme as regras léxicas definidas. Essas regras são geralmente especificadas por expressões regulares que descrevem os padrões válidos na linguagem de programação. O lexer deve ser capaz de distinguir entre diferentes tipos de tokens e ignorar espaços em branco e comentários que não são relevantes para a análise sintática (Isidoro 2016).

A correta implementação do analisador léxico é crucial para a construção de um compilador. Erros na análise léxica podem levar a falhas nas etapas subsequentes de análise sintática e semântica. Portanto, é fundamental que o lexer seja preciso e eficiente na categorização dos tokens.

No contexto deste trabalho, utilizamos a ferramenta Alex para gerar o lexer. Alex permite definir padrões léxicos usando expressões regulares e gera automaticamente o código Haskell correspondente. Essa abordagem facilita a implementação do lexer e garante a precisão na identificação dos tokens.

1.2 Análise Sintática

A análise sintática é a fase do compilador responsável por verificar se uma sequência de tokens gerada pelo lexer segue as regras sintáticas da linguagem de programação. Essa verificação é feita com base em uma gramática formal que define as construções válidas da linguagem.

A análise sintática gera uma estrutura de dados chamada *árvore sintática abstrata* (AST), que organiza as expressões de forma hierárquica, permitindo o processamento das próximas fases de um compilador ou a interpretação por parte de um interpretador.

1.3 Árvore Sintática Abstrata (AST)

A *AST* é uma representação intermediária do programa que captura a estrutura lógica das expressões. Cada nó da árvore representa uma operação ou comando, enquanto os ramos são as expressões que compõem essa operação. A AST é fundamental nos processos subsequentes da compilação e também na interpretação de um programa para a linguagem.

1.4 Interpretação

A interpretação de um programa consiste na execução direta do código-fonte ou de uma representação intermediária, sem que haja a necessidade de uma tradução completa para um código de máquina ou outro formato binário. Um interpretador lê o código, analisa sua estrutura sintática e executa as instruções à medida que as encontra, em tempo de execução.

Ao contrário dos compiladores, que convertem todo o programa em código de máquina antes de executá-lo, o interpretador trabalha diretamente com a árvore sintática abstrata (AST) gerada durante a análise sintática. A função do interpretador é navegar por essa árvore, avaliando expressões, executando comandos e manipulando estruturas de controle, como loops e condicionais, de forma dinâmica.

1.5 Análise Semântica

Enquanto a análise sintática verifica se a sequência de tokens forma estruturas gramaticais válidas conforme a gramática da linguagem, a análise semântica se preocupa em assegurar que essas estruturas façam sentido no contexto da linguagem, seguindo as regras semânticas. Isso inclui verificação de tipos, checagem de declarações de variáveis e funções, verificação de escopo, e outras restrições semânticas da linguagem. Essa fase acontece após a análise sintática

Durante a análise semântica, o compilador utiliza a Árvore Sintática Abstrata (AST) gerada na fase sintática para percorrer o código e validar diversos aspectos, como:

- **Verificação de Tipos**: Garante que as operações realizadas são entre tipos compatíveis, evitando, por exemplo, a soma de um inteiro com um booleano.
- **Resolução de Identificadores**: Verifica se variáveis e funções utilizadas foram previamente declaradas e estão acessíveis no contexto atual.
- **Verificação de Escopo**: Assegura que as regras de escopo da linguagem são respeitadas, impedindo acessos indevidos a variáveis fora de seu contexto.
- **Verificação de Retorno de Funções**: Confirma se as funções retornam valores compatíveis com seus tipos declarados e se todos os caminhos de execução possíveis retornam um valor.
- **Análise de Estruturas de Dados**: Valida a correção de definições de tipos complexos, como registros ou classes, garantindo consistência interna.

A análise semântica é fundamental para garantir que o programa não apenas está sintaticamente correto, mas também que sua execução fará sentido conforme as regras da linguagem. Erros semânticos podem levar a comportamentos inesperados ou falhas em tempo de execução. Portanto, essa fase atua como um filtro para detectar e reportar inconsistências lógicas antes que o código seja interpretado ou compilado para código de máquina.

1.6 Ferramentas utilizadas

1.6.1 Alex

Alex é uma ferramenta para a geração de analisadores léxicos em Haskell. Ele permite a definição de padrões léxicos usando expressões regulares e gera automaticamente o código

Haskell correspondente para a análise desses padrões.

1.6.2 Happy

Happy é uma ferramenta de geração de analisadores sintáticos (parsers) para Haskell, similar ao Yacc usado em C. Ele permite descrever a gramática de uma linguagem e automaticamente gerar o código para a análise sintática. Happy utiliza a LALR(1) para gerar os parsers [1].

1.6.3 Stack

Stack é uma ferramenta de gerenciamento de projetos e construção de pacotes no ecossistema Haskell. Ele foi projetado para facilitar o desenvolvimento de projetos Haskell de maneira eficiente e com builds reprodutíveis. Stack utiliza um sistema de "snapshots", que são coleções de versões específicas de pacotes Haskell, garantindo que os projetos tenham consistência entre diferentes máquinas e ambientes. Isso evita problemas comuns de compatibilidade de versão entre dependências.

Stack também oferece uma integração com o `Stackage`, um repositório curado de pacotes Haskell que assegura que todas as versões dos pacotes dentro de um snapshot sejam compatíveis entre si. Além de gerenciamento de dependências, o Stack inclui ferramentas para compilar, testar, executar e distribuir projetos Haskell, tornando o processo de desenvolvimento mais intuitivo e confiável.

2 Desenvolvimento

Nesta seção, detalhamos o desenvolvimento do projeto. A Subseção 2.1 aborda a estrutura de um programa correto em Lang. A Subseção 2.2 aborda as decisões técnicas e estratégias adotadas para o desenvolvimento do analisador léxico. Na Subseção 2.3, discutimos as escolhas para a implementação do analisador sintático. Na Subseção 2.4, exploramos as decisões relacionadas à construção do interpretador para a linguagem Lang e, por fim, a Subseção 2.5 descreve a implementação da análise semântica.

2.1 Estrutura de um programa em Lang

Para melhor entendimento da implementação deste trabalho como um todo, se faz necessário entender a estrutura de um programa válido em Lang. A estrutura é a que se segue:

- **Program**: Representa o programa como um todo, composto por uma lista de definições (`DefList`).
- **DefList**: É uma lista que pode conter zero ou mais definições (`Def`).
- **Def**: Cada definição pode ser uma função (Func) ou um tipo de registro (`Data`).
- **Data**: Define um tipo de dado registro, permitindo a criação de novos tipos de dados. Um `Data` possui um nome e uma lista de declarações de campos com seus respectivos tipos.

[1] *Happy Docs*. Disponível em: `https://monlih.github.io/happy-docs/`. Acesso em: 07 set. 2024.

- **Func**: Representa uma função, que possui um nome, uma lista de parâmetros (que pode ser vazia), uma lista de tipos de retorno, e uma lista de comandos que compõem o corpo da função. Uma função sem retorno é chamada de procedimento.
- **Command**: São as instruções executáveis da linguagem, como atribuições, condicionais, laço de repetição, chamadas de função, entre outros.
- **Exp**: As expressões que podem ser avaliadas para produzir um valor, incluindo operações aritméticas, lógicas, acesso a variáveis, chamadas de função, etc.

Para que um programa seja considerado válido em Lang, ele deve conter exatamente um procedimento `main`, que serve como ponto de entrada para a execução. Para entendimento completo dos comandos e expressões existentes em Lang, além dos tipos existentes, é disponibilizado dentro da pasta *spec* o arquivo *lang-spec.pdf* com a especificação completa da linguagem.

2.2 Analisador léxico

A implementação do lexer foi baseada em expressões regulares que definem os padrões para cada tipo de token reconhecido pela linguagem Lang. As expressões regulares foram utilizadas para descrever identificadores, tipos, números inteiros, números de ponto flutuante, literais de caracteres, operadores e palavras-chave da linguagem (Manacero 2024).

Como descrito abaixo:

- A expressão regular para identificar números inteiros `integer` foi definida como `$digit+`, onde `$digit` representa qualquer dígito de 0 a 9. O símbolo `+` indica que a sequência deve conter um ou mais dígitos consecutivos.
- Para números de ponto flutuante (`float`), a expressão regular usada foi `$digit+ "."$digit+ | "."$digit+`, que permite reconhecer tanto números na forma "123.456"quanto na forma ".456".
- As expressões regulares `$lowercase`, `$uppercase`, `$alpha`, e `$underscore` foram usadas para definir as categorias básicas de caracteres:
  - `$lowercase` foi definida como `[a-z]`, representando qualquer letra minúscula do alfabeto.
  - `$uppercase` foi definida como `[A-Z]`, representando qualquer letra maiúscula do alfabeto.
  - `$alpha` combina `$lowercase` e `$uppercase`, sendo definida como `[$lowercase $uppercase]`, o que significa que ela reconhece qualquer letra do alfabeto, seja maiúscula ou minúscula.
  - `$underscore` foi definida como _, representando o caractere sublinhado (underscore).
- Os identificadores (`identifier`) foram definidos como uma letra minúscula seguida de uma combinação de letras, dígitos ou underscores (`$lowercase[$alpha $digit $underscore]`).

- Nome de tipos (`typename`) seguem a convenção de começar com uma letra maiúscula, seguida de letras, dígitos ou underscores (`$uppercase[$alpha $digit $underscore]`).
- Literais de caracteres (`@literalChar`) são caracteres únicos delimitados por aspas simples. Esses foram definidos como um único caractere alfabético ou um caractere especial de escape, como (`\n`), (`\t`), (`\b`), (`\r`), (`\'`), ou (`\\`) através da seguinte expressão regular: `@literalChar = $alpha | \\n | \\t | \\b | \\r | \\ | \\\\ | \'` que, quando combinada com a especificação `"'"@literalChar "'"`, forma um caractere literal válido de acordo com a linguagem Lang.

Essas expressões regulares são automaticamente convertidas em autômatos finitos pela ferramenta Alex, que é usada para gerar o código Haskell responsável pela análise léxica. Cada expressão regular corresponde a um estado do autômato, que percorre o código fonte caractere por caractere, categorizando-o conforme as regras definidas.

É importante ressaltar que a definição completa dos tokens para palavras reservadas, comandos, operadores e outros símbolos específicos podem ser encontrados no arquivo Lexer.x fornecido, e não foram detalhadamente especificados neste relatório.

2.3 Analisador sintático

A análise sintática para a linguagem foi desenvolvida em duas grandes partes:

2.3.1 Definição das Estrutura de Dados da AST

No arquivo `Syntax.hs` foi realizado a definição das estruturas de dados que representam a AST da linguagem. Esse arquivo contém a especificação dos tipos de dados para os principais componentes da linguagem, como:

- Program: contém uma lista de definições (DefList).
- DefList: uma lista de definições que podem ser funções ou dados. Ou estar vazia.
- Func: define a estrutura de uma função, com nome, parâmetros, tipo de retorno e comandos.
- Exp: expressões como somas, comparações e chamadas de função.
- Command: comandos como blocos, condicionais, iterações, atribuições e chamadas de função.

Todas as definições podem ser encontradas no arquivo Syntax.hs, acima foram apresentadas apenas algumas.

2.3.2 Criação do arquivo para a ferramenta Happy

O arquivo `Parser.y` contém a implementação do analisador sintático (parser) da linguagem Lang, criado utilizando o Happy. Esse arquivo segue três etapas principais:

- Mapeamento dos tokens gerados pelo lexer (definidos em `Lexer.x`) para o formato esperado pelo Happy.

- Definição da precedência e associatividade dos operadores.
- Definição da estrutura sintática de Lang, descrevendo suas regras de gramática.

Além disso, foi implementada a função `parserError`, que exibe erros de análise sintática, utilizando a função `showToken` para formatar e exibir tokens de forma clara.

2.4 Interpretador

A implementação do interpretador está contida no arquivo `Interpreter.hs`, onde foi definido o ambiente de execução e as funções que realizam a interpretação do código.

2.4.1 Definição do ambiente

No interpretador, criamos um ambiente que consiste em três componentes principais:

- **Ambiente de Variáveis**: Armazena as variáveis em uso, associando nomes a valores.
- **Ambiente de Funções**: Contém as definições das funções disponíveis para chamada durante a execução.
- **Ambiente de Dados**: Guarda as definições dos tipos de registro (`Data`) criados pelo usuário.

O ambiente completo é uma estrutura que encapsula esses três ambientes, permitindo o gerenciamento eficiente das informações necessárias durante a interpretação.

2.4.2 Funções Principais do Interpretador

A função `interpreter` é o ponto de entrada do interpretador. Ela recebe um `Program` e inicia a interpretação, procurando pela função `main` e executando-a.

As principais funções de interpretação são:

- `interpDefList` e `interpDef`: Responsáveis por interpretar a lista de definições e cada definição individualmente. Elas constroem os ambientes de funções e de dados.
- `interpCommandList` e `interpCommand`: Interpretam listas de comandos e comandos individuais, como `print`, chamadas de função, condicionais (`if` e `else`), atribuições, retornos de função, etc.
- `interpExp`: Avalia as expressões, retornando valores que podem ser inteiros, *float*, booleanos, arrays, `null`, caracteres literais ou tipos de registro.
- Funções auxiliares: São utilizadas para apoiar as funções principais na interpretação, como manipulação de l-values, gerenciamento do ambiente e tratamento de valores.

A ideia geral é que, primeiro o interpretador irá interpretar a lista de definições, ou seja, os tipos de registro e funções definidos pelo usuário, armazenando tudo no ambiente. Após isso, será executado o procedimento main, interpretando a lista de comandos e expressões definidas. Se não há um main definido, um erro em tempo de execução é lançado.

Um ponto importante de ressaltar é que as funções para interpretação de comandos, retornam o ambiente modificado, enquanto a interpretação das expressões retornam um valor. Além de poderem lançar erros em tempo de execução.

2.4.3 Decisões de Implementação

Durante o desenvolvimento do interpretador, tomamos algumas decisões importantes que influenciam o comportamento da linguagem:

- **Retorno após os erros**: Quando ocorre um erro durante a interpretação, o interpretador imprime uma mensagem de erro e retorna o valor `null` na interpretação de expressões e o ambiente atual na interpretação de comandos. Além disso, personalizamos algumas mensagens de erro em tempo de execução, como: Index out of bounds quando tenta acessar uma posição não existente no array; Function not found; Unknown type, entre outros. A decisão de personalizar a mensagem de erro nesses casos, foi realizada para facilitar a identificação dos erros durante o desenvolvimento dos exemplos solicitados.
- **Valores Padrão**: Quando uma variável é instanciada com o comando new, ela recebe um valor padrão de acordo com seu tipo:
  - **Int**: 0
  - **Float**: 0.0
  - **Bool**: `false`
  - **Char**: `'a'`
  - **Null**: `null`

  O mesmo vale para variáveis definidas em novos tipos de dados instanciados com Data e para vetores.
- **Entrada de Dados com `read`**: A função `read` aceita apenas valores que pertencem aos tipos primitivos da linguagem: `int`, `char`, `float` e `bool`. Caso o usuário tente ler um valor de outro tipo ou um valor inválido, o interpretador imprime uma mensagem de erro e retorna `null`.
- **Gerenciamento do Ambiente**: Ao chamar uma função, o interpretador cria um novo ambiente local para os parâmetros e variáveis locais da função. Isso isola o escopo da função do ambiente global, evitando conflitos de variáveis. Após a execução da função, o ambiente é restaurado ao estado anterior. Portanto, se passamos uma variável como parâmetro e alteramos o valor dessa variável dentro da função, esse novo valor valerá apenas no escopo da função, não valendo para a variável definida na função ou procedimento que evocou essa função.

2.5 Análise Semântica

A análise semântica foi implementada para garantir que o código fonte da linguagem Lang não só esteja sintaticamente correto, mas também semanticamente válido, respeitando as regras e restrições impostas pela linguagem.

2.5.1 Arquivos Principais

A implementação da análise semântica foi dividida em três arquivos principais:

- **Basics.hs**: Contém definições básicas e funções comuns utilizadas na análise de comandos e expressões. Neste arquivo, são definidos o contexto, os tipos de erros, e estruturas auxiliares.
- **CommandTypeChecker.hs**: Responsável pela verificação semântica dos comandos da linguagem. Este módulo percorre os comandos do programa, verificando declarações, atribuições, chamadas de função e estruturas de controle.
- **ExpTypeChecker.hs**: Encarregado da verificação semântica das expressões. Avalia tipos de expressões aritméticas, booleanas, acesso a variáveis, chamadas de função, entre outras.

2.5.2 Definição do Contexto

Para a análise semântica, foi definido um contexto (`Ctx`) que mantém informações sobre variáveis, funções e tipos de dados definidos no programa. O contexto é dividido em três partes:

- **VarCtx**: Contexto de variáveis, uma lista de pares contendo o nome da variável e seu tipo.
- **FuncCtx**: Contexto de funções, armazenando o nome das funções juntamente com seus tipos de parâmetro e retorno.
- **DataCtx**: Contexto de tipos de dados (`Data`), guardando definições de tipos de registro com seus respectivos campos e tipos.

A definição do contexto em Haskell ficou da seguinte forma:

```
type VarCtx = [(String, Type)]
type FuncCtx = [(String, FuncType)]
type DataCtx = [(String, DataType)]

type Ctx = (VarCtx, FuncCtx, DataCtx)

data FuncType = FuncType [Type] [Type]
    deriving (Eq, Show)

data DataType = DataType [(String, Type)]
    deriving (Eq, Show)
```

2.5.3 Processo de Análise

A análise semântica foi realizada em duas etapas principais:

1. **Coleta de Definições**: O analisador percorre o código fonte, armazenando as definições de funções e tipos de dados no contexto. Isso permite que, ao encontrar uma referência a uma função ou tipo, seja possível verificar se ela foi definida.

2. **Verificação Semântica**: Com o contexto populado, o analisador percorre novamente o código, agora verificando comandos e expressões. Durante essa etapa, são realizadas verificações como:

   - **Verificação de Tipos**: Assegura que operações são realizadas entre tipos compatíveis.
   - **Verificação de Declarações**: Verifica se variáveis e funções utilizadas foram declaradas.
   - **Análise de Escopo**: Garante que variáveis são acessadas dentro de seus escopos válidos.
   - **Tratamento de Retornos de Função**: Confirma se funções retornam os tipos esperados e se o número de valores retornados está correto.

2.5.4 Decisões de Implementação

Uma decisão importante durante a implementação foi a forma de lidar com definições de tipos recursivos. No caso de tipos de dados que referenciam a si mesmos, como em estruturas de árvores ou listas ligadas, foi necessário adicionar o nome do tipo ao contexto antes de processar seus campos. Por exemplo:

```
data Node {
    left :: Node;
    right :: Node;
    value :: Int;
}
```

Para permitir que os campos `left` e `right` referenciem o tipo `Node` dentro da própria definição, adicionamos inicialmente o nome `Node` com uma definição vazia ao contexto de dados. Após processar todos os campos, atualizamos a definição completa no contexto. A ideia dessa implementação veio na tentativa de implementar o exemplo solicitado da árvore binária. Por mais que não conseguimos implementar o exemplo completo, mantemos essa implementação.

3 Execução

Para compilação e execução do programa são fornecidas três opções, como descritas abaixo:

- `stack run -- --untyped "caminho-do-arquivo"`: Este comando irá pular a parte da análise sêmantica, executando o Lexer, Parser e Interpretador.
- `stack run -- --typed "caminho-do-arquivo"`: Com esse comando, a análise semântica será executada antes da interpretação.
- `stack test`: Executa a suíte de testes definida, validando o comportamento esperado do sistema em diferentes cenários de teste. Importante salientar, que o os testes irão apenas executar os programas definidos e não haverá uma comparação de fato com resultados esperados.

Esses comandos devem ser executados dentro da pasta **tp3**.

4 Dificuldades encontradas

De forma geral, a principal dificuldade enfrentada durante o desenvolvimento deste projeto foi o estágio inicial de cada componente. Uma vez entendido a estrutura do que precisa ser realizado, o desenvolvimento se encaminhou bem, apesar de ainda haver dificuldades, principalmente com a linguagem Haskell. Por vezes, sabia-se a lógica da implementação, mas havia a dificuldade em descrevê-la utilizando Haskell.

No parser, percebemos que havia um conflito do tipo shift/reduce, o qual foi resolvido adicionando um comando para especificar que, após o parser reconhecer o token *Command*, ele deve realizar um shift e continuar processando os tokens subsequentes, sem tentar reduzir imediatamente a expressão inteira.

Não conseguimos implementar os programas de árvore binária e de grafos devido ao tempo escasso. O de árvore binária teve seu desenvolvimento iniciado, alguns ajustes foram realizados na interpretação e análise semântica durante seu desenvolvimento, mas não foi finalizado.

A Tabela 1 apresenta os exemplos solicitados e se foram implementados ou não.

Tabela 1: Implementação dos exemplos solicitados

Exercício	**Implementado**
Cálculo de fatorial	Sim
Cálculo do n-ésimo termo da sequência de Fibonnacci	Sim
Implementação de um algoritmo de ordenação de arrays	Sim
Implementação de uma pilha e de suas operações (criação de uma pilha vazia, empilhar, desempilhar, número de elementos)	Sim
Implementação de uma árvore binária e de suas operações (criação de uma árvore vazia, inserir, pesquisar, remover)	Não
Implementação de algoritmos sobre grafos (busca em profundidade, busca em largura e caminho mínimo)	Não

5 Referências

Referências

[Isi16] John Isidoro. *Análise Lexical*. Accessed: 2024-08-16. 2016. URL: `https://johnidm.gitbooks.io/compiladores-para-humanos/content/part1/lexical-analysis.html`.

[Man24] Aleardo Manacero. *Capítulo 2: Análise Lexical*. Acessado em: 16 de agosto de 2024. 2024. URL: `https://www.dcce.ibilce.unesp.br/~aleardo/cursos/compila/cap02.pdf`.