UniRuy & Área 1 | Wyden

PROGRAMA DE ENGENHARIA DA COMPUTAÇÃO

TEORIA DE COMPILADORES

JOÃO VICTOR DE DEUS MARTINS

Teoria de Compiladores: LINGUAGENS DE PROGRAMAÇÃO

Salvador - Bahia - Brasil

2022

JOÃO VICTOR DE DEUS MARTINS

Teoria de Compiladores: LINGUAGENS DE PROGRAMAÇÃO

Trabalho Acadêmico elaborado junto ao programa de Engenharia UniRuy & Área 1 | Wyden, como requisito para obtenção de nota parcial da AV1 na disciplina Teoria de Compiladores no curso de Graduação em Engenharia da Computação, que tem como objetivo consolidar os tópicos do plano de ensino da disciplina.

Orientador: Prof. MSc. Heleno Cardoso

Salvador - Bahia - Brasil
2022

TERMO DE APROVAÇÃO

JOÃO VICTOR DE DEUS MARTINS

TEORIA DE COMPILADORES: LINGUAGENS DE PROGRAMAÇÃO

Trabalho Acadêmico aprovado como requisito para obtenção de nota parcial da AV1 na disciplina Teoria de Compiladores, UniRuy & Área 1 | Wyden, pela seguinte banca examinadora:

BANCA EXAMINADORA

Profº. MScº. Heleno Cardoso
Wyden

Salvador, 05 de Outubro de 2022

Dedico este trabalho acadêmico a todos que contribuíram direta ou indiretamente com minha formação acadêmica.

Agradecimentos

Primeiramente agradeço a Deus. Ele, sabe de todas as coisas, e através da sua infinita misericórdia, se fez presente em todos os momentos dessa trajetória, concedendo-me forças e saúde para continuar perseverante na minha caminhada.

E a todos aqueles que contribuíram direta ou indiretamente para a minha formação acadêmica.

"A educação tem raízes amargas, mas os seus frutos são doces".

Aristóteles.

Resumo

Ao longo desta resenha, os fundamentos teóricos das linguagens de programação são desenvolvidos e as questões pragmáticas envolvidas em sua implementação são explicadas. Há muitos fatores a serem considerados ao projetar uma linguagem de programação. Aqui mostramos aos compiladores e tradutores o papel central da teoria dos tipos e da semântica operacional para ajudar a definir conceitos de linguagem e entender suas propriedades.

Palavras-chaves: Evolução, Padrões, Tradutores, Compiladores.

Abstract

Throughout this review, the theoretical foundations of programming languages are developed and the pragmatic issues involved in their implementation are explained. There are many factors to consider when designing a programming language. Here we show compilers and translators the central role of type theory and operational semantics in helping to define language concepts and understand their properties.

Keywords: Evolution, Standards, Translators, Compilers.

Lista de abreviaturas e siglas

TOKEN	- Componente léxico.
RAM	- Random Access Memory.

Sumário

[1] De acordo com a Associação Brasileira de Normas Técnicas. NBR 6023.

1 LINGUAGENS DE PROGRAMAÇÃO

1.1 Introdução

O objetivo desta resenha é expor e desmistificar alguns detalhes e princípios subjacentes de como as tecnologias de Linguagens de programação funcionam, desde a sua evolução até temas mais complexos como compiladores a fim de demonstrar o potencial das máquinas universais de computação e sistemas que estão em execução em nossos desktops.

1.2 Execução/Método

Resenha desenvolvida através de analises de artigos científicos apurados.

1.2.1 Repositório de Pesquisa

IEEE

1.2.2 String de Busca por Repositório

"Compilers Programming Languages"
"Programming Languages History Compilers"
"Programming LanguagesStructure Compilers"

1.2.3 Artigos Selecionados

Associations as a language construct.
Object-oriented programming as the end of history in programming languages.
Task specific programming languages as a first programming language.
Foundations of Probabilistic Logic Programming: Languages, Semantics, Inference and Learning.

1.2.4 Resenha dos Artigos Selecionados

Desde que Charles Babbage inventou o mecanismo de diferença em 1822, os computadores precisavam de uma maneira de direcioná-los para realizar tarefas específicas. Isso significa que é chamado de linguagem de programação. As linguagens de computador consistem principalmente em uma série de etapas que conectam um determinado programa; estas se tornam uma série de etapas que são digitadas no computador e depois executadas; posteriormente, essas linguagens adquiriram recursos avançados, como ramificação lógica e orientação a objetos. As linguagens de computador dos últimos cinquenta anos foram divididas em duas fases, a primeira linguagem principal e a segunda linguagem principal, ambas em uso hoje. As primeiras linguagens de programação desenvolvidas na década de 1950 eram mais amigáveis aos humanos. É linguagem assembly mnemônica. Primeiro, as instruções em linguagem assembly são mnemônicos para instruções de máquina. Mais tarde, macros foram adicionadas à linguagem assembly, o que ajudou os programadores a definir abreviações parametrizadas para sequências de instruções de máquina usadas com frequência. A primeira linguagem de programação rudimentar de alto nível foi o FORTRAN, desenvolvido pela IBM na segunda metade da década de 1950, para computação científica. FORTRAN foi a primeira linguagem de programação de alto nível com implementações de funções. Mais tarde, na década de 1950, o COBOL foi desenvolvido por um comitê de curto alcance. Foi desenvolvido para trabalhar com dados de negócios. Em 1958, John McCarthy desenvolveu o LISP. Foi desenvolvido para computação simbólica. Hoje, existem muitas linguagens de programação disponíveis no mercado. As linguagens de programação podem ser divididas de diferentes maneiras. Uma dessas formas de categorizar linguagens de programação é geracional. A linguagem de máquina é a linguagem de primeira geração, a linguagem assembly é a linguagem de segunda geração e as linguagens de programação de alto nível como C, C++, LISP, FORTRAN e JAVA são a linguagem de terceira geração. A linguagem de quarta geração é NOMAD para relatórios, SQL para bancos de dados e postscript para formatação. Prolog e OPS5 são exemplos de linguagens de quinta geração. Outra forma de classificação é essencial para a linguagem. Ele específica como a computação deve ser feita e é tão declarativa da linguagem quanto especifica o que a computação deve ser feita. C, C++, C, JAVA são exemplos de linguagens imperativas. Uma linguagem funcional é considerada uma linguagem declarativa. ML, Haskel e restrições baseadas em lógica são linguagens declarativas. Métodos computacionais as linguagens de programação baseadas

na arquitetura von Neumann são consideradas linguagens von Neumann. FORTRAN e C são a língua von Neumann. Uma linguagem de programação que consiste em programação orientada a objetos, de classe, objeto e suporte é considerada uma linguagem orientada a objetos. Linguagem como C++, C, ruby, java é línguas orientadas a objetos.

Nem todas as linguagens de programação populares têm documentação, e as linguagens ainda podem existir e ser populares por anos sem uma descrição formal. Uma linguagem pode ter uma ou mais implementações, esses comportamentos atuam como um padrão de fato, sem que esse comportamento seja formalmente documentado. Perl é um exemplo de linguagem que não é especificada, enquanto o PHP só foi especificado em 2014, após 20 anos de uso. O design e a implementação de uma linguagem podem ocorrer simultaneamente ou as ordens podem ser revertidas. Essa é uma prática comum hoje. Isso ocorre porque especificações e implementações estão em comunicação umas com as outras, as implementações devem descrever precisamente o comportamento de uma implementação, enquanto as especificações devem ser práticas, consistentes e possíveis. A prática de escrever uma especificação antes de uma implementação foi evitada nas últimas décadas devido às complicações associadas a esta última. No entanto, as linguagens ainda são ocasionalmente implementadas e se tornam populares sem uma especificação formal: uma implementação é necessária para uso, enquanto uma especificação é benéfica, mas não necessária (informalmente, "conversas de código"). Primeiro (e possivelmente último) a ter uma definição formal antes de ser implementado foi o ALGOL 68.

Um tradutor muda um programa de um idioma para outro. Eles são um processador de linguagem de programação. Um tradutor de código de programa para máquina requer um programa original para inspiração. Suas descobertas traduzem o programa com erros mínimos. Linguagens de alto nível como C, C++ e Java são traduzidas para uma linguagem de baixo nível chamada de programa objeto ou programa de máquina por um compilador. Um compilador implementa várias fases para converter linguagem superior para linguagem inferior. Um fluxo de caracteres de um cliente passa por essas fases para chegar ao idioma de destino. Um pré-processador é um programa que opera sob as diretivas das linhas de comando do pré-processador, Um Assembler é um tradutor que converte um programa em linguagem assembly em um programa em linguagem de computador. Em comparação, os 0s e 1s de um computador são mais difíceis de ler e escrever devido à sua natureza bruta. Por causa do Assembler', escrever programas é mais fácil e o programa traduzido é mais amigável do que um computador com 0's e 1's.

A estrutura de um compilador Um compilador deve efetivamente traduzir a linguagem escrita infinita do programa em uma linguagem de saída final. Isso requer a compreensão dos princípios e estratégias para resolver problemas relacionados ao processo de compilação. Basicamente temos duas fases de compiladores, a fase de Análise e Síntese.

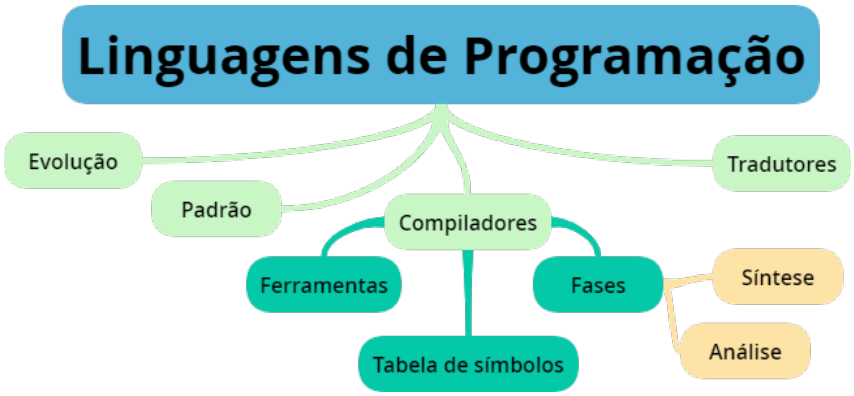
A fase de análise lê o programa de origem e o divide em tokens e cria uma representação intermediária do programa de origem. Ele verifica e aponta os programas de origem quanto a erros sintáticos e semânticos, reúne informações sobre o programa fonte e prepara a tabela de símbolos. A tabela de símbolos será usada durante todo o processo de construção. Isso também é conhecido como o front-end do compilador.

Na fase de síntese ele pegará a entrada do estágio de análise (representação intermediária e tabela de símbolos) e produzirá o código de nível de máquina de destino. Isso também é chamado de backend do compilador. Falando sobre a Tabela de Símbolos, é uma estrutura de dados utilizada e mantida pelo compilador, composta por todos os nomes de identificadores e seus tipos. Ele ajuda o compilador a funcionar sem problemas, encontrando rapidamente os identificadores. Qualquer pessoa que escreve software usa ferramentas semelhantes, como editor de linguagem, depurador, criadores de perfil, equipamento de teste e assim por diante. Ferramentas dedicadas para cada fase do compilador foram criadas usando uma linguagem específica. Essas ferramentas usam componentes e algoritmos exclusivos que só existem nessa linguagem. Abaixo estão listadas ferramentas comuns usadas para compilar a construção: Parser Generator: Gera um analisador baseado na gramática de uma linguagem de programação. Scanner Generator: Gera um lexer a partir de expressões regulares marcadas em linguagens de programação. Mecanismo de tradução baseado em sintaxe: Gera um conjunto de rotinas para árvore de análise e geração de código intermediário. Gerador de código: Gera um gerador de código a partir de um conjunto de regras que convertem cada operação intermediária em código de máquina para a máquina de destino. Data Flow Analysis Engine: Reúne informações sobre como os valores são transferidos de uma parte do programa para outra. Compiler Build Kit: Fornece um conjunto de regras de composição para construir os vários estágios do compilador.

Detectar e relatar erros em programas fonte é a principal função de um compilador. Erros podem ocorrer em qualquer estágio da compilação. Um bom compilador deve determinar exatamente o número da linha do programa onde ocorreu o erro. Os vários erros que podem ocorrer em diferentes níveis de compilação são os seguintes: O primeiro são os erros lexicais (scanners), alguns dos tipos mais comuns aqui incluem caracteres

ilegais ou não reconhecidos, causados principalmente por erros de digitação. Uma maneira comum disso acontecer é o programador digitar um caractere que é ilegal na linguagem de qualquer maneira e nunca usado. Outro tipo de erro que o scanner pode detectar são constantes de caracteres ou strings não terminadas. Isso acontece sempre que um programador digita aspas e se esquece de acompanhá-las. O segundo tipo de erros são erros de sintaxe, esses erros são capturados pelo analisador. Esses erros são os mais comuns. A parte difícil é onde continuar analisando depois de encontrar um erro. Se o analisador não for escrito com cuidado ou se o esquema de detecção e recuperação de erros for insuficiente, o analisador encontrará um erro e depois disso, dando falsas mensagens de erro pelo resto do programa. Se ocorrer um erro, o que você deseja que aconteça é que o compilador ignore quaisquer tokens incorretos e continue capturando o erro sem gerar uma mensagem de erro que não seja um erro, mas o resultado do primeiro erro. Este aspecto é tão importante que alguns compiladores são classificados de acordo com quão bons ou ruins são seus sistemas de detecção de erros. O terceiro erro é a semântica, as semânticas usadas nas linguagens de computador são muito mais simples do que as usadas na linguagem falada. Isso ocorre porque em linguagens de computador, tudo é muito explícito. Um possível erro semântico em um programa tem a ver com o fato de que algumas instruções podem ser sintaticamente corretas, mas não têm significado, e nenhum código pode ser gerado para alcançar o significado da instrução. Erros do tipo IV podem ser encontrados durante a otimização do código, durante a análise do fluxo de controle, pode haver algumas instruções que nunca podem ser alcançadas. Um quinto tipo de erro pode ocorrer durante a geração do código, a arquitetura do computador também desempenha um papel importante na geração do código. Uma sexta categoria de erros pode ser encontrada quando o compilador tenta criar uma entrada na tabela de símbolos, nesta categoria, pode haver um identificador com várias declarações com propriedades conflitantes.

Mapa Mental



1.2.5 Perguntas e Respostas

1. Qual a última linguagem de programação a possuir uma definição formal antes da sua implementação?

ALGOL 68

2. Quais as fases do compilador descrevam-nas?

Basicamente temos duas fases de compiladores, a fase de Análise e Síntese. Análise Ele reúne informações sobre o programa fonte e prepara a tabela de símbolos. A tabela de símbolos será usada durante todo o processo de construção. Isso também é conhecido como o front-end do compilador. Síntese Ele pegará a entrada do estágio de análise (representação intermediária e tabela de símbolos) e produzirá o código de nível de máquina de destino. Isso também é chamado de backend do compilador.

1.3 Conclusão

As linguagens de programação nos permitem realizar uma ampla gama de tarefas de processamento de dados. A partir de uma compreensão básica de programação, os usuários podem realizar atos aparentemente mágicos com arquivos e RAM. Os programas de computador são a maneira mais confiável de transmitir nossas instruções à máquina. O uso do código nos permite ser precisos e expressivos ao mesmo tempo em que fornece um registro completo de nossas ações. Além disso, a codificação é fácil para outros replicarem.

Referências[1]

OSTERBYE, K. Associations as a language construct. ago. 6DC.

JVAN EMDEN, M. H. Object-oriented programming as the end of history in programming languages. ago. 6DC.

GGUZDIAL, M.; MCCRACKEN, W. M.; ELLIOTT, A. Task specific programming languages as a first programming language. ago. 6DC.

RIGUZZI, F. Foundations of Probabilistic Logic Programming: Languages, Semantics, Inference and Learning. [s.d.].

[1] De acordo com a Associação Brasileira de Normas Técnicas. NBR 6023.