Compiladores

CIÊNCIA DA COMPUTAÇÃO

Prof. Dr. João Paulo Aramuni

UNIVERSIDADE FUMEC

Sumário

* **Análise Semântica**
  * Introdução
  * Tabela de símbolos
  * Alocação de memória
  * Análise semântica em tipos de variáveis
  * Análise semântica em comandos

Análise Semântica

* **Introdução**
  * Até o momento vimos as etapas de análise léxica, que quebra o programa fonte em **tokens**, e de análise sintática, que valida as regras de sintaxe da linguagem de programação e gera uma **árvore sintática**.
  * Agora veremos a etapa de análise semântica.
    * Algumas literaturas tratam a análise semântica como "ações semânticas" realizadas pelo compilador durante a fase de geração de código intermediário.
    * Cabe ao projetista de compilador escolher se esta fase será feita separadamente ou junto à geração de código intermediário.

Análise Semântica

* **Introdução**
  * Por que o compilador precisa da fase de análise semântica?
    * Existem determinadas regras referentes à linguagem que **NÃO** são possíveis de serem representadas com expressões regulares/gramáticas regulares (fase léxica) ou com gramáticas livres de contexto (fase sintática).
    * Por exemplo:
      * Todo identificador deve ser declarado antes de ser utilizado.
      * Não é possível representar a regra acima através de uma gramática.

Análise Semântica

* **Introdução**
  * Muitas verificações devem ser realizadas com meta-informações e com elementos que estão presentes em vários pontos do código fonte, distantes uns dos outros.

  * O analisador semântico utiliza a árvore sintática (resultado da análise sintática) e a tabela de símbolos para fazer a análise semântica.

Análise Semântica

* **Introdução**
  * A análise semântica é responsável por verificar aspectos relacionados ao significado (sentido) das instruções.
  * Essa é a terceira etapa do processo de compilação. Nesse momento ocorre a validação de uma série de regras que não podem ser verificadas nas etapas anteriores.
  * É importante ressaltar que muitos dos erros semânticos tem origem de regras dependentes da linguagem de programação. Por esse motivo a análise semântica irá variar MUITO de linguagem para linguagem.

Análise Semântica

* **Introdução**
  * As atividades de tradução executadas pelos compiladores baseiam-se fundamentalmente em uma perfeita compreensão da semântica da linguagem a ser compilada, uma vez que é disso que depende a construção dos gerados de código intermediário, responsáveis pela obtenção do código-objeto a partir do programa fonte.

Análise Semântica

* **Introdução**
  * Em muitos compiladores, a geração de código intermediário vem acompanhada das atividades de análise semântica, responsáveis pela captação do significado exato das sentenças da linguagem de programação que estão contidas no texto-fonte.

Análise Semântica

* **Introdução**
  * Em muitos textos encontrados na literatura, dá-se genericamente o nome de *ações semânticas* a essa classe de tarefas executadas pelo compilador.

  * Esse nome deve ser entendido com cautela, pois nesse grupo de atividades costumam ser englobadas, na prática, não apenas ações autenticamente semânticas, mas também toda sorte de operações adicionais que sejam necessárias à compilação, mas não compreendidas no escopo das análises léxica e sintática.

Análise Semântica

* **Introdução**
  * A palavra semântica deveria referir-se ao exato significado com que o compilador deveria interpretar a sentença em análise no contexto em que ela foi encontrada no programa-fonte.

  * Estendendo-se tal conceito a todos os programas que se podem denotar em uma linguagem de programação, a semântica dessa linguagem pode ser entendida como sendo o conjunto de interpretações que se podem atribuir a todos esses programas.

Análise Semântica

* **Introdução**
  * Ao contrário da sintaxe, que é de fácil formalização com a ajuda de metalinguagens de baixa complexidade, a semântica, embora também possa ser formalmente expressa, infelizmente exige para isso notações significativamente mais complexas, de interpretação mais difícil, sendo, em geral, expressa por simbologias densas e poucos legíveis.

Análise Semântica

* **Introdução**
  * Assim sendo, da observação dos relatórios de especificação da maioria das linguagens de programação, pode-se constatar que, em geral, a semântica costuma ser definida informalmente, muitas vezes apenas por exemplos e textos explicativos informais, em linguagem natural.

  * Não é trivial a tarefa de descrever completamente uma linguagem de programação, ainda que de pequeno porte e complexidade, de tal modo que tanto sua sintaxe quanto sua semântica estejam completas e precisamente contidas nessa descrição.

Análise Semântica

* **Introdução**
  * As validações que não podem ser executadas pelas etapas anteriores devem ser executadas durante a análise semântica a fim de garantir que o programa fonte esteja coerente e o mesmo possa ser convertido para linguagem intermediária.

  * A análise semântica percorre a árvore sintática e relaciona os identificadores com seus dependentes de acordo com a estrutura hierárquica.

Análise Semântica

* **Introdução**
  * Essa etapa captura informações sobre o programa fonte para que a fase subsequente gere o código objeto.

  * Uma das tarefas mais importantes da análise semântica é a verificação de tipos.
    * Neste momento, o compilador verifica se cada operador recebe os operandos permitidos e especificados na linguagem fonte.

Análise Semântica

* **Introdução**
  * Um exemplo que ilustra muito bem essa etapa de validação de tipos é a atribuição de objetos de tipos ou classes diferentes.

  * Em alguns casos, o compilador realiza a conversão automática de um tipo para outro que seja adequado à aplicação do operador. Por exemplo a expressão:

```
var s: String;
s := 2 + '2';
```

Análise Semântica

* **Introdução**
  * Exemplos de tarefas do analisador semântico:
    * Verificar se os identificadores são declarados antes de serem usados;
    * Verificar se os tipos são compatíveis em uma operação;
    * Verificar se os identificadores foram utilizados dentro do escopo ao qual foram declarados.
    * Verificar se o número de parâmetros em uma chamada de função está correto em relação ao que foi declarado.

Análise Semântica

* **Introdução**
  * Considere o exemplo abaixo:

$$i = a + b;$$

  * Quais questões o analisador semântico deve considerar para validar a expressão acima?

Análise Semântica

* **Introdução**
  * Considere o exemplo abaixo:

$$i = a + b;$$

  * O identificador i foi declarado?
  * O identificador i é uma variável?
  * Qual o escopo da variável i?
  * Qual é o tipo da variável i?
  * O tipo da variável i é compatível com os demais identificadores e operadores?

Análise Semântica

* **Introdução**
  * Os tipos de dados são muito importantes nessa etapa de compilação, eles são como notações que as linguagens de programação utilizam para representar um conjunto de valores.

  * Com base nos tipos o analisador semântico pode definir quais valores podem ser manipulados, isso é conhecido como type checking.

Análise Semântica

* **Introdução**
  * Algumas linguagens utilizam um mecanismo muito interessante chamado "inferência de tipos", que permite a uma variável assumir vários tipos durante o seu ciclo de vida.

  * Isto permite que a variável possa ter vários valores. Nesses casos o compilador infere o tipo da variável em tempo de execução, esse tipo de mecanismo está diretamente relacionado ao mecanismo de *Generics* do Java. A validação de tipos passa a ser realizada em tempo de execução.

Análise Semântica

* **Introdução**
  * Tipicamente, as **ações semânticas** englobam funções tais como as que estão comentadas a seguir:

Análise Semântica

* **Criação e manutenção de tabelas de símbolos**
  * Em geral, essa tarefa é executada pelo analisador léxico, porém, em muitos compiladores, por razão de reaproveitamento dos programas de análise léxica e mesmo sintática, os respectivos analisadores são construídos de modo tal que suas funções se limitam estritamente às operações mínimas a que se referem.

  * Assim, todas as atividades secundárias que usualmente são executadas por esses módulos são transferidas para outra parte do compilador (geralmente na forma de ações semânticas), à qual se transfere a responsabilidade de executá-las.

Análise Semântica

* **Associação entre os símbolos e seus atributos**
  * Conhece-se como tabela de símbolos a coleção dos identificadores encontrados pelo compilador ao longo da sua inspeção do texto-fonte.
  * Identificadores isolados, porém, não conseguem dar ao compilador todas as informações de que necessita acerca das características e propriedades dos objetos aos quais dão nome.
  * Dessa maneira, torna-se necessário associar ao nome de cada um deles um conjunto de atributos, que contenham informações suficientes para individualizar o elemento da linguagem a que o nome se refere, indicando, conforme a conveniência, características adicionais de tal elemento que sejam necessárias no processo de geração de código intermediário.

Análise Semântica

* **Manutenção de informações sobre o escopo dos identificadores**
  * Conforme a linguagem de programação a que se destina, o compilador deve se encarregar da resolução do escopo dos elementos da linguagem, ou seja, de determinar a localidade de tais objetos, problema esse muito ligado ao comportamento do programa em tempo de execução.

  * Para que o compilador possa conhecer a relação que existe entre cada componente declarado e o escopo a que pertence, dando-lhes a correta interpretação em todos os pontos do texto-fonte, é preciso que a tabela de símbolos registre a associação existente entre cada identificador registrado e o correspondente escopo.

Análise Semântica

* **Manutenção de informações sobre o escopo dos identificadores**
  * Em alguns compiladores, o próprio analisador sintático se incumbe de identificar as fronteiras entre os diversos escopos existentes no texto-fonte, e tal informação é passada ao analisador léxico para que ele registre e utilize a informação. Em outros, a execução de tais operações é promovida por ações semânticas especificamente associadas à identificadores das fronteiras do programa-fonte nas quais ocorrem mudanças de escopo.

Análise Semântica

* **Representação dos diversos tipos de dados**
  * Linguagens modernas proporcionam ao programador um grande repertório de tipos para os dados que permitem manipular, em geral representados por agregados homogêneos (vetores, matrizes, tabelas) ou heterogêneos (estruturas).

  * Além de disponibilizarem declarações e tipos de dados nativos, muitas dessas linguagens costumam permitir ao programador a especificação de novos tipos de dados personalizados, não nativos da linguagem.

  * Nesse caso, cabe ao compilador a tarefa de registrar as especificações dos diversos tipos de dados que o programa utiliza, bem como de utilizar tais informações para prever a ocupação de memória, em tempo de execução, por parte dos objetos que forem declarados como sendo do tipo especificado.

Análise Semântica

* **Análise das restrições de uso dos identificadores**
  * Cabe ao compilador, por meio de suas ações semânticas, efetuar a verificação da coerência de utilização de cada identificador nas diversas situações em que é referenciado no texto-fonte. Tal informação se encontra disponível na tabela de símbolos, usualmente na forma de um conjunto de atributos, associados a cada identificador presente na tabela.

  * Assim, o compilador deve, para cada identificador que for encontrado durante a análise do programa-fonte, verificar se estão ou não registrados para ele atributos compatíveis com o contexto no qual o identificador foi utilizado no programa, reportando mensagens de erro sempre que for detectada alguma incompatibilidade.

Análise Semântica

* **Verificação do escopo dos identificadores**
  * Mediante consulta à informação do escopo no qual um identificador está sendo referenciado, o compilador deve executar procedimentos capazes de garantir que todos os identificadores utilizados no texto-fonte correspondam a objetos devidamente definidos nos pontos em que forem referenciados.

  * Entre os principais pontos a serem investigados, destaca-se a, pertinência desse identificador ao conjunto dos identificadores corretamente definidos no escopo corrente e a compatibilidade dos atributos associados a tais identificadores com o uso que deles é feito em cada ocorrência.

Análise Semântica

* **Identificação de declarações contextuais**
  * Enquanto muitas linguagens de programação associam tipos aos identificadores mediante declarações explícitas, outras permitem que essa associação seja feita de modo implícito, a partir da observação do contexto em que o identificador é encontrado no programa.

  * É também função das ações semânticas do compilador localizar tais identificadores em seu contexto sintático, inferir os atributos que lhes cabem e associar-lhes tais atributos na tabela de símbolos.

Análise Semântica

* **Identificação de declarações contextuais**
  * Um caso particular dessa característica se manifesta no caso de certas linguagens que apresentam identificadores predefinidos, com semântica fixa, e que fazem parte da própria linguagem de programação.

  * Situação semelhante ocorre com relação a variáveis e outros objetos, que, embora possam ter seus identificadores escolhidos pelo programador, apresentam valor inicial preestabelecido (default) pela linguagem ou então pelo compilador específico que a realiza.

Análise Semântica

* **Verificação da compatibilidade de tipos (type-checking)**
  * Às ações semânticas cabe ainda a importante função de verificar a coerência do uso dos identificadores referenciados no programa, de forma tal que se garanta que os elementos da linguagem por eles simbolizados, que representam os dados do programa nos diversos comandos em que são empregados, se refiram a dados compatíveis com os tipos a eles designados nas respectivas declarações.

  * Essa verificação (type-checking) constitui um valioso recursos auxiliar para as atividades ligadas à geração de código intermediário e se aplica à maioria das linguagens que permitem, em determinados contextos (por exemplo, em expressões), a utilização mista de dados de tipos diferentes.

Análise Semântica

* **Verificação da compatibilidade de tipos (type-checking)**
  * Embora notacionalmente corrente, a mistura de tipos impõe ao gerador de código intermediário o teste das condições particulares associadas a cada caso para que possa construir um código-objeto que inclua todas as conversões que forem eventualmente necessárias à correta realização das operações especificadas.

Análise Semântica

* **Gerenciamento de memória – considerações gerais**
  * Quanto às ações de gestão da memória, o ambiente de execução deve se ocupar de controlar o uso do espaço de endereços livres de memória, alocado ao programa-objeto pelo compilador.
  * É possível classificar os requisitos de memória de um programa em duas categorias: os requisitos estáticos e os dinâmicos.
  * Requisitos estáticos de memória referem-se a necessidades que o programa tem de espaço de trabalho, que independam dos dados do programa, podendo ser totalmente resolvidos em tempo de compilação.

* **Gerenciamento de memória – considerações gerais**
  * Já os requisitos dinâmicos relacionam-se a necessidades do programa, que variam em função dos dados e da execução e surgem durante sua operação, não podendo ser antecipados para o tempo de compilação.

  * Diversos requisitos dinâmicos de memória de um programa decorrem da execução de chamadas de procedimentos recursivos, do uso de agregados com limites dinâmicos, do gerenciamento de memória de trabalho (heap) e do emprego de ambientes de execução que agreguem variáveis declaradas de diferentes escopos, especialmente no caso de linguagens estruturadas em blocos.

Análise Semântica

* **Alocação estática de memória**
  * O compilador deve incorporar mecanismos pelos quais sejam previstos, calculados e controlados os requisitos de memória do programa que está sendo traduzido.
  * Por razões de eficiência, deve ser dada preferência aos métodos que promovam, sempre que possível, as atividades de alocação preditiva, pois, quando, quando viáveis, elas podem ser executadas precocemente.
  * Em muitas situações, os requisitos de memória do programa-objeto podem ser antecipados com precisão durante a compilação e, nesse caso, as correspondentes atividades de alocação de memória podem ser integralmente efetuadas antes que o programa seja posto em execução.

Análise Semântica

* **Alocação estática de memória**
  * Linguagens de baixa complexidade costumam empregar elementos componentes cujas exigências de memória são previsíveis: todos os dados têm dimensões fixas e seus procedimentos, não recursivos, ocupam uma área constante de memória.

  * Em casos como esse, a maneira mais econômica de gerenciar a memória é por meio de uma política de gerenciamento conhecida pelo nome de alocação estática de memória, cuja utilização em tempo de execução dispensa a presença de qualquer software de apoio.

  * Nela, faz-se uma alocação estática e contínua dos elementos do programa, com base em um dimensionamento prévio das áreas de dados temporárias, bem como dos espaços de memória necessários para o código e para os endereços de retorno e a passagem de parâmetros.

Análise Semântica

* **Alocação automática de memória**
  * Em um segundo tipo de organização de memória, conhecida como alocação automática, os movimentos de alteração do tamanho da pilha ficam estritamente associados à ocasião da mudança de escopo que ocorre à entrada e à saída dos blocos ou procedimentos.

  * Linguagens com estrutura de blocos, que permitem declarações de variáveis locais, como é o caso da maior parte das linguagens atualmente em uso, disponibilizam ao programador, por sua vez, uma série de recursos com exigências dinâmicas de memória, apenas viáveis com o apoio de um ambiente de execução apropriado.

  * Isso se dá porque as dimensões dos elementos que compõem a área de dados (por exemplo, matrizes de dimensões variáveis, áreas de pilha para os argumentos de procedimentos recursivos) dos programas escritos nessas linguagens dependem dos dados, cujos valores raramente são conhecidos em tempo de compilação.

Análise Semântica

* **Alocação automática de memória**
  * A organização de dados imposta em linguagens com estruturas de blocos exige, em tempo de execução, que se adote uma disciplina compatível de alocação para os dados declarados nos diversos blocos do programa.
  * Em tempo de execução, a cada ingresso em um bloco ou procedimento, a pilha é automaticamente alterada, pois nela devem ser inseridos registros de ativação, áreas especiais de dados que representam a dinâmica de funcionamento da estrutura de blocos do programa em operação.
  * Esses descritores compreendem uma área para representar os objetos declarados no procedimento, outra para seus parâmetros e ainda outra, referente a informações de controle.

Análise Semântica

* **Alocação automática de memória**
  * Entre as informações de controle presentes em um registro de ativação, destacam-se: o endereço de retorno, apontadores para outros registros de ativação acessíveis no ponto de ativação do bloco e valores de retorno do procedimento.

  * Convém lembrar que procedimentos recursivos ou reentrantes podem apresentar simultaneamente mais de um registro de ativação e cada um deles identifica e personaliza uma diferente instância ativa do procedimento.

Análise Semântica

* **Alocação dinâmica de memória**
  * Um terceiro tipo de gerenciamento de memória, conhecido como alocação dinâmica, é exigido por algumas linguagens nas quais as necessidades de memória nem podem ser previstas em tempo de compilação, nem resolvidas com o auxílio de uma pilha.

  * Cria-se, nesse caso, uma área de rascunho (heap) na qual, de acordo com as características do programa, e na medida de suas estritas necessidades, vão sendo alocadas áreas para os componentes do programa.

  * De forma análoga ao que acontece na alocação, ao final da utilização de cada uma dessas áreas de memória, o espaço por elas ocupado é devolvido ao ambiente de execução para que possa ser reciclado.

Análise Semântica

* **Alocação dinâmica de memória**
  * Um exemplo de elementos do programa com requisitos de alocação de memória dinâmica são as cadeias de caracteres (strings) de comprimento arbitrário, ou objetos que possam mudar dinamicamente de dimensão.

  * Outra situação que exige tal recurso é encontrada em muitas linguagens modernas, as quais, durante a execução de um programa, permitem que ele requisite ou libere áreas arbitrárias da heap.

  * O gerenciamento de memória dinâmica não é trivial, devido à variação da dimensão entre os diversos elementos nela armazenados e ao fato de ser também arbitrário o instante em que é liberada pelo programa cada uma das porções de memória ocupadas pelos seus dados.

Análise Semântica

* **Alocação dinâmica de memória**
  * Pode assim instalar-se com facilidade o fenômeno da fragmentação, em que áreas livres não contíguas se dispersam pela memória, as quais, na forma dispersa em que se encontram, tendem a ser de mais difícil reaproveitamento.

  * O cenário assim estabelecido exige a aplicação da popular estratégia de recuperação conhecida como garbage collection, que envolve a compactação de memória mediante a movimentação física de áreas ocupadas pelos dados, com a finalidade de mover as áreas livres para a mesma região, formando assim uma única grande área livre.

  * O procedimento de garbage collection promove a compactação de memória sempre que, em resposta a novas requisições, não for possível alocar dinamicamente as regiões disponíveis de memória, em virtude do excesso de fragmentação e da indisponibilidade de áreas vagas contíguas capazes de satisfazer os requisitos da solicitação.

Análise Semântica

* **Representação do ambiente de execução dos procedimentos**
  * Outra tarefa que cabe ao compilador, por meio das ações semânticas, é a de interpretar corretamente o contexto de execução dos procedimentos declarados no programa.
  * Para tanto, é muito importante o esquema adotado para a passagem de parâmetros (por nome, por valor, por referência etc.), a maneira pela qual essas transferências são realizadas e a forma como os resultados de execução dos procedimentos são retornados ao contexto chamador.
  * Outro elemento a ser considerado nesse item é a comunicação eventual com ambientes externos (outros programas, bibliotecas, sistema operacional, módulos compilados separadamente, rotinas de pacotes de aplicação etc.), nos quais os protocolos de comunicação eventualmente exigidos devem ser executados pelos mecanismos de compilação, aos cuidados das rotinas semânticas.

Análise Semântica

* **Tradução do programa para a linguagem-objeto**
  * A principal função das ações semânticas é exatamente criar, a partir do texto-fonte, com base nas informações tabeladas e nas saídas dos outros analisadores, uma interpretação desse texto-fonte expressa em notação adequada.

  * Essa notação não precisa ser obrigatoriamente uma linguagem de máquina específica, sendo com frequência representada por alguma linguagem intermediária adotada no processo de compilação.

Análise Semântica

* **A construção de um ambiente de execução**
  * Em geral, o programa traduzido não é autônomo, exigindo o apoio de programas auxiliares, usados como elementos de suporte para sua execução.
  * Esse ambiente de execução é geralmente composto de recursos computacionais responsáveis por atividades que, na linguagem de saída do compilador, não costumam estar disponíveis como operações nativas e se realiza na forma de uma biblioteca de execução, de uma interface com o usuário e de outra com o sistema operacional.
  * Quanto mais rico o ambiente de execução, mais compacto tende a ser o código-objeto gerado, pois muitas funções que, de outra maneira, teriam de ser explicitamente geradas no código-objeto, podem ser indiretamente executadas pela requisição dos serviços correspondentes disponíveis no ambiente de execução.

Análise Semântica

* **A construção de um ambiente de execução**
  * Algumas linguagens exigem, adicionalmente, mecanismos permanentes de gerenciamento de memória que suportam as características dinâmicas compatíveis com os tipos de dados disponibilizados pela linguagem.
  * Outras, por seu caráter interativo, necessitam de um ambiente em que a comunicação com o operador é significativa e que, por essa razão, se realiza de maneira mais complexa, às vezes com o auxílio de sofisticadas interfaces com o usuário.
  * Em alguns casos, costumam ser incorporados aos ambientes de execução ferramentas e instrumentos de medida pelos quais podem ser feitos levantamentos de desempenho, mediante métricas de avaliação, ou acompanhamentos da trajetória da execução do programa, para efeito de testes e de depuração.

Análise Semântica

* **Comunicação entre ambientes de execução**
  * Adicionalmente, pode ser necessário incluir, em tempo de execução, um mecanismo pelo qual informações possam ser passadas entre dois ou mais ambientes de execução.

  * A finalidade de tal recurso é permitir que procedimentos se comuniquem entre si (pela passagem de parâmetros), que trechos de programa pertencentes a escopos diferentes, porém que possam coexistir, acessem objetos aos quais possuam direitos de acesso e que processos paralelos (tarefas) sejam realizados, comunicando-se por mensagens e áreas compartilhadas.

  * Cabe aos procedimentos criados pelas ações semânticas ter um controle preciso de tais mecanismos e gerar um código que permita utilizá-los adequadamente em todos os casos de interesse.

Análise Semântica

* **Geração de código intermediário**
  * Com base na tradução prévia para uma linguagem intermediária, o código-objeto do programa pode finalmente ser construído.
  * Em geral, a geração desse código pode ser feita de duas maneiras: sem cuidados adicionais, ou gerando-se um código com características que favoreçam eventuais operações subsequentes de otimização.
  * No primeiro caso, o código geralmente constitui uma interpretação literal do programa-fonte e, portanto, sua qualidade tende a mostrar-se muito inferior à de um código equivalente produzido por um programador humano. No segundo caso, o código é construído em um formato compatível para ser tratado por uma seção de otimização disponível no compilador.
  * Quase sempre, o código-objeto assim gerado é relocável, para que seja possível liga-lo facilmente a outros códigos, desenvolvidos separadamente.

Análise Semântica

* **Otimização do código-objeto**
  * Os compiladores modernos, mesmo aqueles desenvolvidos para máquinas de pequeno porte, cada vez mais exploram as técnicas de otimização com a finalidade de construir de forma automática um código de baixo nível de qualidade comparável ao código produzido por bons programadores, incorporando para isso diversas técnicas de otimização ao compilador.

  * A primeira dessas técnicas corresponde à otimização independente de máquina, que efetua manipulações da árvore sintática com a finalidade de reduzi-la, compactá-la e eliminar redundâncias e o consequente excesso desnecessário de processamento.

  * Essa classe de melhorias envolve a otimização de expressões, a eliminação de subexpressões comuns, a fatoração do cálculo de subexpressões de uso frequente, as otimizações de construções iterativas etc.

Análise Semântica

* **Otimização do código-objeto**
  * Outra técnica de aperfeiçoamentos do código refere-se às otimizações dependentes de máquina, para as quais o hardware em que se dá a execução do programa influi na otimização a ser realizada.
  * Entre outras, podem ser incluídas nessa classe a otimização do uso de registradores, a otimização do uso do conjunto de instruções da máquina, a redução do tempo de execução do programa pela escolha adequada de sequências eficientes de instruções de baixo nível etc.
  * Muitos programas de otimização dependente de máquina operam por meio de sucessivas transformações diretas de um código-objeto disponível, de qualidade inferior, previamente gerado, identificando padrões de código e substituindo-os por padrões equivalentes conhecidos, mais eficientes.

Análise Semântica

* **Tabela de símbolos**
  * A análise semântica, assim como as demais fases, necessita coletar informações presentes na tabela de símbolos.
  * Como foi visto, uma vez extraídos do texto-fonte pelo analisador léxico, os identificadores são coletados em uma tabela e a cada um dos elementos armazenados nessa tabela de símbolos é associada uma única identificação.

Análise Semântica

* **Tabela de símbolos**
  * Cada instância de um identificador presente no texto do programa-fonte promove uma consulta no compilador, com eventual alteração ou atualização da tabela de símbolos.

  * Assim, toda vez que um identificador é localizado em uma declaração no texto-fonte, ou em algum contexto em que esteja de fato sendo declarado, caso tal identificador já não conste na tabela de símbolos, deve ser inserido e marcado como um símbolo definido; caso contrário, naturalmente, uma mensagem de erro deve ser reportada.

Análise Semântica

* **Tabela de símbolos**
  * Encontrados em outros contextos, os identificadores devem ser interpretados como referências aos elementos da linguagem a eles associados, definidos no texto-fonte do programa por meio de declarações.
  * Assim sendo, em todas essas situações, a tabela de símbolos deve ser consultada, em busca do símbolo em questão.

Análise Semântica

* **Tabela de símbolos**
  * Se o símbolo consta na tabela, acompanhado de um atributo que o indica como tendo sido previamente definido, nada mais é feito além de uma consulta a seus atributos.

  * Eventualmente, pode ser oportuno marcá-lo com um atributo que o indique como referenciado, ou registrar a informação da posição do programa na qual o símbolo foi referenciado, com a finalidade de coletar dados para a construção de tabelas de referências cruzadas.

Análise Semântica

* **Tabela de símbolos**
  * Se o símbolo não consta na tabela, isso geralmente configura uma situação de erro, exceto nos casos de compiladores de linguagens que admitem referências a objetos declarados adiante (para esses casos, é preferível optar por um esquema de compilação em mais de um passo).

  * A criação e manutenção da tabela de símbolos são operações vitais para o funcionamento do compilador, pois essa tabela é a estrutura de dados mais importante no tratamento dos aspectos semânticos da compilação.

Análise Semântica

* **Tabela de símbolos**
  * Na tabela de símbolos são guardados os nomes dos objetos definidos pelo programador e é através dela que eles são referenciados simbolicamente, por nome, ao longo de todo o programa.

  * Nos casos mais simples, as tabelas de símbolos são meros vetores linearmente organizados a partir de elementos de comprimento fixo, com política de acesso aleatório para leitura e crescimento pela extremidade mais recente (fila).

Análise Semântica

* **Tabela de símbolos**
  * Em algumas situações, a linguagem-fonte permite que os identificadores tenham um comprimento máximo muito grande, impedindo, por razões práticas, sua construção na forma de uma tabela de elementos homogêneos em comprimento.
  * Para esses casos, uma estruturação na forma de listas de símbolos pode resolver esse problema, dando maior flexibilidade à tabela.

Análise Semântica

* **Tabela de símbolos**
  * A principal aplicação prática desse tipo de tabelas se dá no caso de linguagens com escopo único e, nesse caso, a coleta do identificador e seu armazenamento na tabela podem ser realizados diretamente pelo analisador léxico.

  * Entretanto, em certas situações, convém remover do analisador léxico essa atividade e introduzi-la como parte das rotinas semânticas.

Análise Semântica

* **Tabela de símbolos**
  * Assim, obtém-se um compilador mais claro, pois, dessa forma, todas as atividades ligadas à tabela de símbolos podem ser agrupadas, em vez de dispersadas entre os vários componentes do compilador.

  * No entanto, as informações referentes aos identificadores necessárias à compilação não se restringem apenas aos nomes dos elementos da linguagem, pois a cada nome costumam estar associados diversos complementos, que funcionam como seus atributos.

Análise Semântica

* **Tabela de símbolos**
  * Para memorizar tais complementos e associá-los aos identificadores, uma estrutura de dados adicional deve ser mantida: a tabela de atributos.

  * Uma organização trivial, que permite a coexistência das tabelas de símbolos e de atributos na mesma estrutura de dados, armazena os atributos em uma tabela (vetor) cujos elementos correspondem posicionalmente aos elementos da tabela de símbolos, qualquer que seja sua ordem.

Análise Semântica

* **Tabela de símbolos**
  * Essa organização peca, no entanto, pelo fato de os atributos e os identificadores a que se referem serem mantidos distantes, obscurecendo sua interpretação.

  * Como alternativa imediata, as tabelas de símbolos podem ser fisicamente organizadas como listas ligadas, nas quais cada elemento da lista possa conter não apenas símbolos de comprimento variável, mas também atributos a eles associados.

Análise Semântica

* **Tabela de símbolos**
  * Há compiladores que utilizam, para armazenar as tabelas, uma estrutura de dados com política de acesso de pilha, que apresenta a vantagem de se tornar disponível para uso em outras aplicações simultâneas, tais como a construção da árvore abstrata do programa.

  * As diversas estruturas de dados ficam, nesse caso, fisicamente embaralhadas, porém, se realizadas na forma de listas ligadas, podem permanecer logicamente separadas entre si pela limitação aos acessos impostos pelos apontadores que as realizam.

Análise Semântica

* **Tabela de símbolos**
  * O armazenamento em pilha das estruturas de dados que materializam a tabela de símbolos dá margem ao uso desse tipo de organização em compiladores de linguagens que apresentam mais de um escopo estático.

  * Para tais linguagens, diferentemente do que ocorre com linguagens de escopo único, um mesmo identificador pode ser declarado diversas vezes no programa, desde que em escopos separados.

Análise Semântica

* **Tabela de símbolos**
  * Em compiladores de um único passo, o desenvolvimento da tabela de símbolos acompanha a estrutura estática do programa-fonte.
  * Sendo de um único passo, após efetuar a análise de um trecho do texto-fonte, tais compiladores não mais voltam a examinar suas partes já analisadas.

Análise Semântica

* **Tabela de símbolos**
  * Dessa forma, uma vez terminada a análise de um trecho de programa correspondente a um dos escopos definidos por ele, todos os símbolos que nele tenham sido porventura declarados deixam de ser utilizados, podendo ser removidos da tabela de símbolos.

  * Adicionalmente, o fato de um ponto qualquer do programa poder fazer parte de mais de um escopo torna possível que algum identificador venha a ser utilizado simultaneamente em diversos escopos.

Análise Semântica

* **Tabela de símbolos**
  * Isso sugere a utilização múltipla dos identificadores registrados na tabela de símbolos, de modo que o nome dos objetos indique, por meio de um de seus atributos, o identificador que a ele corresponde em cada caso.

  * Dessa maneira, o identificador fica instanciado uma única vez na tabela, sendo daí por diante compartilhado, se necessário, por mais de um elemento da tabela de atributos.

Análise Semântica

* **Tabela de símbolos**
  * O uso de uma pilha na realização de tal arranjo possibilita, nesse caso, a coexistência de três estruturas: os identificadores, os respectivos atributos e a árvore abstrata.

  * É necessário, ainda, que sejam marcados, na tabela ou fora dela, limites que indiquem a região da tabela em que se encontram todos os identificadores pertencentes a cada escopo declarado no programa.

Análise Semântica

* **Tabela de símbolos**
  * Dessa forma, terminada a análise do trecho do programa associado a um dado escopo, os identificadores compreendidos nesse escopo podem ser mais facilmente removidos da tabela.

  * Nessas circunstâncias, em cada etapa da compilação, a tabela de símbolos e atributos conterá estritamente as informações relativas aos objetos declarados no escopo a que se refere o elemento que estiver sendo tratado na ocasião.

Análise Semântica

* **Tabela de símbolos**
  * Assim, os demais identificadores utilizados no programa são automaticamente desprezados na análise, quer por terem sido previamente descartados, quer por ainda não terem sido inseridos na tabela.

  * Em compiladores organizados em mais de um passo, a situação muda radicalmente, já que se torna necessário preservar toda a informação coletada para ser utilizada em passos posteriores da compilação.

Análise Semântica

* **Tabela de símbolos**
  * Por essa razão, a estrutura de dados que representa a tabela de símbolos deve refletir preferencialmente a estrutura do programa-fonte a partir do qual é construída.
  * Assim sendo, em um passo preliminar da compilação, pode ser construída uma árvore que reflita estruturalmente o encadeamento estático dos escopos definidos pelo programa-fonte.

Análise Semântica

* **Tabela de símbolos**
  * Para uso nos passos seguintes de compilação, só seria necessário, a rigor, memorizar os atributos associados aos identificadores, mas alguns compiladores preservam também os nomes originais.

  * Tais nomes são utilizados para a emissão de mensagens relativas aos objetos a que se referem os atributos, não em formato codificado, mas na forma simbólica original, utilizada pelo programador.

Análise Semântica

* **Alocação de memória**
  * Um compilador precisa implementar com precisão as abstrações incorporadas na definição da linguagem fonte.
  * Essas abstrações, tipicamente, incluem os conceitos que discutimos anteriormente, como nomes, escopos, associações (do inglês, *bindings*, amarrações ou ligações), tipos de dados, operadores, procedimentos, parâmetros e construções de fluxo de controle.

Análise Semântica

* **Alocação de memória**
  * O compilador precisa cooperar com o sistema operacional e outros softwares do sistema para dar suporte a essas abstrações na máquina alvo.

  * Para fazer isso, o compilador cria e gerencia um ambiente em tempo de execução no qual assume que seus programas objeto estão sendo executados.

Análise Semântica

* **Alocação de memória**
  * Esse ambiente trata de uma série de questões, tais como o layout e a alocação de endereços de memória para os objetos nomeados no programa fonte, os mecanismos usados pelo programa objeto para acessar variáveis, as ligações entre os procedimentos, os mecanismos para passagem de parâmetros e as interfaces para o sistema operacional, dispositivos de entrada e/ou saída, e outros programas.

Análise Semântica

* **Alocação de memória**
  * Iremos abordar os temas de “alocação de endereços de memória” e o “acesso a variáveis e dados”.
  * Veremos o gerenciamento de memória com alguns detalhes, incluindo a alocação de pilha, o gerenciamento da heap e a coleta de lixo.

Análise Semântica

* **Alocação de memória**
  * Organização da memória
    * Do ponto de vista do projetista do compilador, o programa objeto é executado em seu próprio espaço de endereçamento lógico, no qual cada valor no programa possui um endereço.
    * O gerenciamento e a organização desse espaço de endereçamento lógico são compartilhados entre o compilador, o sistema operacional e a máquina alvo. O sistema operacional mapeia os endereços lógicos em endereços físicos, que usualmente se espalham pela memória.

Análise Semântica

* **Alocação de memória**
  * Organização da memória
    * Conforme sabemos, a quantidade de memória necessária para um nome é determinada a partir do seu tipo.

    * Um tipo de dado básico, como um caractere, um número inteiro ou de ponto flutuante, pode ser armazenado em um número inteiro de bytes. A área de memória para um tipo agregado, como um arranjo ou uma estrutura, precisa ser grande o suficiente para conter todos os seus componentes.

Análise Semântica

* **Alocação de memória**
  * Organização da memória
    * O layout de memória para os objetos de dados é fortemente influenciado pelas restrições de endereçamento da máquina alvo.
    * Em muitas arquiteturas, as instruções para somar números inteiros podem esperar que os inteiros estejam alinhados, ou seja, colocados em um endereço divisível por 4.

Análise Semântica

* **Alocação de memória**
  * Organização da memória
    * Embora um arranjo de dez caracteres só precise de bytes suficientes para manter dez caracteres, um compilador pode alocar 12 bytes para obter o alinhamento apropriado, deixando 2 bytes sem uso.
    * O espaço não usado em razão das considerações de alinhamento é conhecido como espaço vazio (*padding*).

Análise Semântica

* **Alocação de memória**
  * Organização da memória
    * Quando há pouco espaço, um compilador pode compactar os dados para que nenhum espaço vazio seja deixado; nesse caso, podem ser necessárias instruções adicionais durante a execução para posicionar os dados compactados de modo que eles possam ser operados como se estivessem corretamente alinhados.

Análise Semântica

* **Alocação de memória**
  * Organização da memória
    * Muitas linguagens de programação permitem que o programador aloque e libere dados sob controle do programa.

    * Por exemplo, C possui as funções **malloc** e **free**, que podem ser usadas para obter e liberar blocos arbitrários de espaço de memória. A heap é usada para gerenciar esse tipo de dados de longa duração.

Análise Semântica

* **Alocação de memória**
  * Alocação de memória estática versus dinâmica
    * O layout e a alocação de dados para endereços de memória em tempo de execução são questões fundamentais no gerenciamento de memória.
    * São questões delicadas porque o mesmo nome em um texto de programa pode referir-se a múltiplas localizações em tempo de execução.

Análise Semântica

* **Alocação de memória**
  * Alocação de memória estática versus dinâmica
    * Os dois adjetivos, estática e dinâmica, distinguem entre tempo de compilação e tempo de execução, respectivamente.
    * Dizemos que a decisão de alocação de memória é estática se puder ser feita pelo compilador examinando apenas o texto do programa, e não o que o programa faz quando é executado.

Análise Semântica

* **Alocação de memória**
  * Alocação de memória estática versus dinâmica
    * Reciprocamente, a decisão é dinâmica se só puder ser decidida enquanto o programa estiver executando.
    * Muitos compiladores utilizam alguma combinação das duas estratégias a seguir para a alocação dinâmica de memória:

Análise Semântica

* **Alocação de memória**
  * Alocação de memória estática versus dinâmica
    * 1) Memória de **pilha:** Nomes locais a um procedimento têm espaço alocado em uma pilha. A pilha admite a política normal de chamada e retorno de procedimentos.
    * 2) Memória **heap:** Os dados que podem sobreviver à chamada do procedimento que os criou usualmente são alocados em um heap de memória reutilizável. O heap é uma área da memória virtual que permite que objetos ou outros elementos de dados obtenham memória quando criados e retornem essa memória quando invalidados.

Análise Semântica

* **Alocação de memória**
  * Alocação de memória estática versus dinâmica
    * Para dar suporte ao gerenciamento de heap, a 'coleta de lixo' permite que o sistema de execução detecte elementos de dados inúteis e reutilize sua memória, mesmo que o programador não retorne seu espaço explicitamente.
    * A coleta de lixo automática é um recurso essencial de muitas linguagens modernas, apesar de ser uma operação difícil de ser feita eficientemente; em algumas linguagens, ela pode nem ao menos ser possível.

Análise Semântica

* **Alocação de memória**
  * Alocação de espaço na pilha
    * Quase todos os compiladores para as linguagens que utilizam procedimentos, funções ou métodos como unidades de ações definidas pelo usuário gerenciam pelo menos parte de sua memória em tempo de execução como uma pilha.

    * Toda vez que um procedimento é chamado, o espaço para suas variáveis locais é colocado em uma pilha, e, quando o procedimento termina, esse espaço é retirado da pilha.

Análise Semântica

* **Alocação de memória**
  * Alocação de espaço na pilha
    * Essa organização não apenas permite que o espaço seja compartilhado pelas chamadas de procedimento cujas durações não se sobrepõem no tempo, mas também nos permite compilar código para um procedimento de modo que os endereços relativos de suas variáveis não locais sejam sempre iguais, independentemente da sequência de chamadas de procedimento.

Análise Semântica

* **Alocação de memória**
  * Dados de tamanho variável na pilha
    * O sistema de gerenciamento de memória em tempo de execução frequentemente precisa tratar a alocação de espaço para objetos cujos tamanhos não são conhecidos durante a compilação, mas os quais são locais a um procedimento e, assim, podem ser alocados na pilha.

    * Em linguagens modernas, os objetos cujos tamanhos não podem ser determinados em tempo de compilação são alocados no espaço do heap.

Análise Semântica

* **Alocação de memória**
  * Dados de tamanho variável na pilha
    * Contudo, também é possível alocar objetos, arranjos ou outras estruturas de tamanho desconhecido na pilha.

    * A razão para preferir colocar objetos na pilha, se possível, é que evitamos o custo da coleta de lixo em seu espaço. Observe que a pilha só pode ser usada para um objeto se ele for local a um procedimento e o objeto fica inacessível quando o procedimento retorna.

Análise Semântica

* **Alocação de memória**
  * Gerenciamento do heap
    * O heap é uma porção de memória usada para dados que residem indefinidamente, ou até que o programa os exclua explicitamente.
    * Enquanto as variáveis locais tipicamente se tornam inacessíveis quando seus procedimentos terminam, muitas linguagens nos permitem criar objetos ou outros dados, cuja existência não esteja ligada à ativação do procedimento pelo qual são criados.

Análise Semântica

* **Alocação de memória**
  * Gerenciamento do heap
    * Por exemplo, tanto C++ quanto Java disponibilizam para o programador a palavra reservada **new** a fim de criar objetos que podem ser passados – ou para os quais podem ser passados apontadores – de um procedimento para outro, de modo que continuam a existir muito depois que o procedimento que os criou termine.

    * Esses objetos são armazenados em um **heap**.

Análise Semântica

* **Alocação de memória**
  * Gerenciamento do heap
    * Precisamos entender agora como funciona o gerenciamento de memória, o **subsistema** que aloca e libera espaço dentro do heap; ele serve como uma interface entre os programas de aplicação e o sistema operacional.
    * Para linguagens como C ou C++, que liberam porções de memória manualmente (por instruções explícitas do programa, como free ou delete), o gerenciador de memória também é responsável por implementar a liberação.

Análise Semântica

* **Alocação de memória**
  * Gerenciador de memória
    * O gerenciador de memória administra todo o espaço livre na memória heap o tempo inteiro. Ele realiza duas funções básicas:

    * 1) Alocação: Quando um programa solicita memória para uma variável ou objeto, o gerenciador de memória disponibiliza uma porção contígua de memória heap com o tamanho solicitado. Se for possível, ele satisfaz uma solicitação de alocação usando o espaço livre no heap. Se nenhuma porção do tamanho necessário estiver disponível, ele procura aumentar o espaço de memória do heap obtendo bytes consecutivos de memória virtual do sistema operacional. Se o espaço estiver esgotado, o gerenciador de memória passa essa informação de volta ao programa de aplicação.

Análise Semântica

* **Alocação de memória**
  * Gerenciador de memória
    * O gerenciador de memória administra todo o espaço livre na memória heap o tempo inteiro. Ele realiza duas funções básicas:

    * 2) Liberação: O gerenciador de memória retorna o espaço liberado ao repositório de espaço livre, de modo que este espaço possa ser reusado para satisfazer outras solicitações de alocação. Os gerenciamentos de memória tipicamente não retornam memória ao sistema operacional, mesmo que o uso do heap do programa diminua.

Análise Semântica

* **Alocação de memória**
  * Gerenciador de memória
    * O gerenciamento de memória seria mais simples se (a) todas as solicitações de alocação fossem para porções do mesmo tamanho, e (b) a memória fosse liberada previsivelmente, digamos, o primeiro alocado, o primeiro liberado.
    * Existem algumas linguagens, como Lisp, para as quais a condição (a) é verdadeira; a linguagem Lisp pura utiliza somente um elemento de dado a partir do qual todas as estruturas de dados são construídas.

Análise Semântica

* **Alocação de memória**
  * Gerenciador de memória
    * A condição (b) também é verdadeira em algumas situações, sendo a mais comum a de dados que podem ser alocados na pilha de execução.

    * Contudo, na maioria das linguagens, nem (a) nem (b) em geral são verdadeiros. Em vez disso, elementos de dados de tamanho diferentes são alocados, e não existe uma boa maneira de prever os tempos de vida de todos os objetos alocados.

Análise Semântica

* **Alocação de memória**
  * Gerenciador de memória
    * Assim, o gerenciador de memória deve estar preparado para atender, em qualquer ordem, solicitações de alocação e liberação de qualquer tamanho, variando desde um byte até todo o espaço de endereços do programa.

    * As propriedades desejáveis em gerenciadores de memória são:

Análise Semântica

* **Alocação de memória**
  * Gerenciador de memória
    * 1) Eficiência de espaço. Um gerenciador de memória deverá minimizar o espaço total do heap necessário por um programa. Isso permite que programas maiores sejam executados em um espaço de endereço virtual fixo. A eficiência do espaço é alcançada minimizando-se a 'fragmentação'.

Análise Semântica

* **Alocação de memória**
  * Gerenciador de memória
    * 2) Eficiência do programa. Um gerenciador de memória deverá fazer bom uso do subsistema de memória para permitir que os programas sejam executados mais rapidamente. Conforme veremos mais adiante, o tempo gasto para executar uma instrução pode variar bastante, dependendo de onde os objetos são colocados na memória.

Análise Semântica

* **Alocação de memória**
  * Gerenciador de memória
    * 2) Felizmente, os programas tendem a exibir 'localidade' um fenômeno que se refere à maneira agrupada não aleatoriamente, na qual os programas típicos acessam a memória. Administrando com atenção o posicionamento de objetos na memória, o gerenciador de memória pode utilizar melhor o espaço, e, possivelmente, fazer com que o programa execute mais rapidamente.

Análise Semântica

* **Alocação de memória**
  * Gerenciador de memória
    * 3) Baixo custo. Como as alocações e liberações de memória são operações frequentes em muitos programas, é importante que sejam o mais eficiente possível, ou seja, queremos minimizar o custo – a fração do tempo de execução gasta realizando alocação e liberação. Observe que o custo das alocações é dominado por pequenas solicitações; o custo de gerenciamento de objetos grandes é menos importante, pois usualmente pode ser amortizado por uma quantidade maior de computação.

Análise Semântica

* **Alocação de memória**
  * Localidade em programas
    * A maioria dos programas exibe um alto grau de localidade; ou seja, gastam a maior parte do seu tempo executando uma fração relativamente pequeno do código e usando apenas uma pequena fração dos dados.

Análise Semântica

* **Alocação de memória**
  * Localidade em programas
    * Dizemos que um programa tem localidade **temporal** se for provável que os endereços de memória que ele acessa sejam acessados novamente dentro de um curto período de tempo.
    * Dizemos que um programa possui localidade **espacial** se for provável que os endereços de memória vizinhos do endereço acessado também sejam acessados dentro de um curto período de tempo.

Análise Semântica

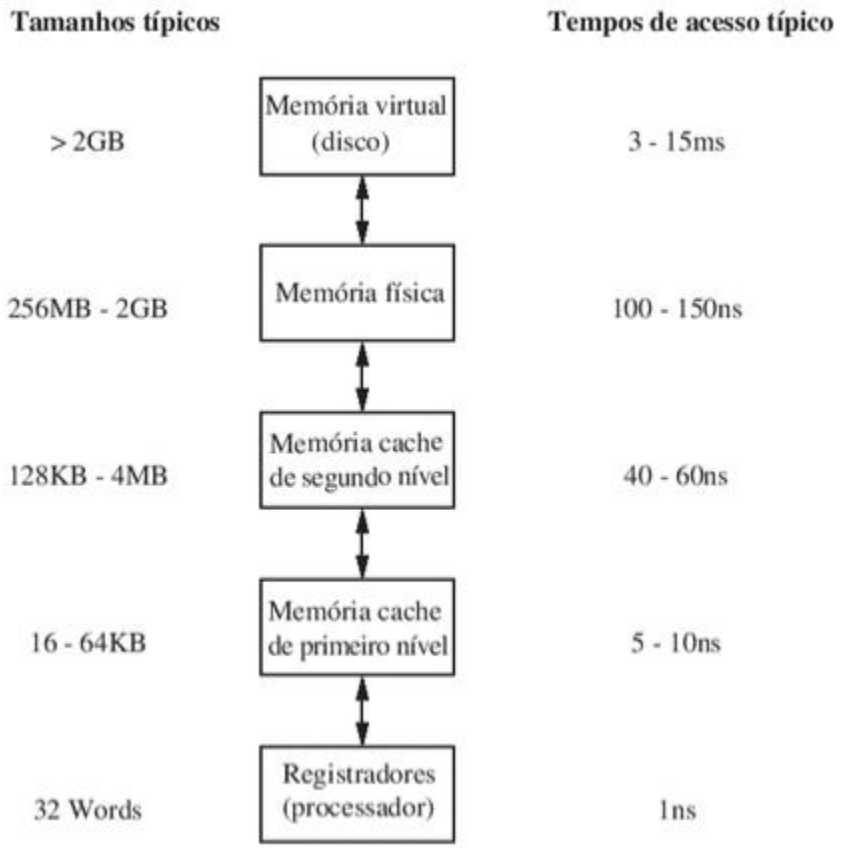
* **Alocação de memória**
  * Localidade em programas
    * A sabedoria convencional diz que os programas gastam 90% do seu tempo executando 10% do código. Saiba por quê:

    * 1) Os programas frequentemente contêm muitas instruções que nunca são executadas. Os programas construídos com componentes e bibliotecas utilizam apenas uma pequena fração da funcionalidade fornecida. Além disso, quando os requisitos mudam e os programas evoluem, os sistemas legados frequentemente contêm muitas instruções que não são mais usadas.

Análise Semântica

* **Alocação de memória**
  * Localidade em programas
    * A sabedoria convencional diz que os programas gastam 90% do seu tempo executando 10% do código. Saiba por quê:
    * 2) Apenas uma pequena fração do código que poderia ser invocado é realmente executada em uma rodada típica do programa. Por exemplo, as instruções para tratar entradas ilegais e casos excepcionais, embora críticas para a exatidão do programa, raramente são invocadas em alguma execução particular.
    * 3) O programa típico gasta a maior parte do seu tempo executando loops mais internos e ciclos recursivos pequenos em um programa.

- **Alocação de memória**
  - Localidade em programas
    - A localidade nos permite tirar proveito da hierarquia de memória de um computador moderno, como mostra a Fig. 1, abaixo.



Configurações típicas de hierarquia de memória.

Análise Semântica

* **Alocação de memória**
  * Localidade em programas
    * Colocando a maioria das instruções e dados comuns na memória rápida, porém pequena, e deixando o restante na memória lenta, porém grande, podemos reduzir significativamente o tempo médio de acesso à memória de um programa.

Análise Semântica

* **Alocação de memória**
  * Localidade em programas
    * Descobriu-se que muitos programas exibem localidade temporal e espacial no modo como acessam instruções e dados. Contudo, os padrões de acesso a dados geralmente mostram uma variância maior do que os padrões de acesso a instruções.

    * Políticas como a de manter os dados usados mais recentemente na hierarquia mais rápida funcionam bem para programas comuns, mas podem não funcionar bem para alguns programa com uso intensivo de dados – por exemplo, aqueles que percorrem arranjos muito grandes em ciclos.

Análise Semântica

* **Alocação de memória**
  * Localidade em programas
    * Frequentemente, apenas examinando o código, não sabemos quais de suas seções serão mais utilizadas, especialmente para determinada entrada.
    * Mesmo que saibamos quais instruções serão muito executadas, a memória cache mais rápida em geral não é grande o bastante para conter todas elas ao mesmo tempo. Portanto, temos de ajustar dinamicamente o conteúdo da memória mais rápida e usá-la para conter instruções que, provavelmente, serão usadas bastante no futuro próximo.

Análise Semântica

* **Alocação de memória**
  * Otimização usando a hierarquia de memória
    * A política de manter as instruções mais usadas na cache tende a funcionar bem; em outras palavras, o passado geralmente é uma boa previsão do uso futuro da memória.
    * Quando uma nova instrução é executada, há alta probabilidade de que a próxima instrução também seja executada. Esse fenômeno é um exemplo da localidade espacial.

Análise Semântica

* **Alocação de memória**
  * Otimização usando a hierarquia de memória
    * Uma técnica efetiva para melhorar a localidade espacial das instruções é fazer com que o compilador coloque os blocos básicos (sequências de instruções que sempre são executadas sequencialmente) que provavelmente seguirão uns aos outros consecutivamente – na mesma página, ou até na mesma linha de cache, se possível.

    * As instruções pertencentes ao mesmo loop ou à mesma função também têm alta probabilidade de serem executadas juntas.

Análise Semântica

* **Alocação de memória**
  * Otimização usando a hierarquia de memória
    * Também podemos melhorar a localidade temporal e espacial dos acessos aos dados em um programa alterando o layout dos dados ou a ordem da computação.
    * Por exemplo, programas que visitam grandes quantidades de dados repetidamente, realizando cada vez uma pequena quantidade de computação, não têm um bom desempenho.

Análise Semântica

* **Alocação de memória**
  * Otimização usando a hierarquia de memória
    * É melhor se pudermos trazer alguns dados de um nível mais lento da hierarquia da memória para um nível mais rápido (por exemplo, do disco para a memória principal) uma vez, e realizar todas as computações necessárias sobre esses dados enquanto eles residem no nível mais rápido.
    * Esse conceito pode ser aplicado recursivamente para reusar dados na memória física, nas memórias cache e nos registradores.

Análise Semântica

* **Alocação de memória**
  * Coleta de lixo
    * Os dados que não podem ser referenciados geralmente são conhecidos como lixo. Muitas linguagens de programação de alto nível removem o peso do gerenciamento de memória manual do programador, oferecendo a coleta de lixo automática, a qual libera dados inalcançáveis.
    * A coleta de lixo existe desde a implementação inicial da linguagem Lisp em 1958. Outras linguagens significativas que oferecem coleta de lixo incluem Java, Perl, ML, Modula-3, Prolog e Smalltalk.

Análise Semântica

* **Alocação de memória**
  * Coleta de lixo
    * A coleta de lixo é a reivindicação de porções de memória contendo objetos que não podem mais ser acessados por um programa.
    * Precisamos assumir que os objetos têm um tipo que pode ser determinado pelo coletor de lixo em tempo de execução. A partir da informação de tipo, podemos saber o tamanho do objeto e quais de seus componentes contêm referências (apontadores) a outros objetos.

Análise Semântica

* **Alocação de memória**
  * Coleta de lixo
    * Um programa de usuário, modifica a coleção de objetos no heap. O programa cria objetos adquirindo espaço do gerenciador de memória, e pode introduzir e descartar referências a objetos existentes.
    * Os objetos tornam-se lixo quando o programa não puder alcança-los. O coletor de lixo encontra esses objetos não alcançáveis e reivindica seu espaço entregando-os ao gerenciador de memória, que administra o espaço livre.

Análise Semântica

* **Alocação de memória**
  * Coleta de lixo
    * Nem todas as linguagens são boas candidatas à coleta de lixo automática. Para que um coletor de lixo funcione, ele precisa ser capaz de dizer se qualquer elemento de dados indicado ou componente de um elemento de dados é ou poderia ser usado como um apontador para uma porção de memória alocada.

    * Uma linguagem em que o tipo de qualquer componente de dados pode ser determinado é considerada como sendo **segura quanto ao tipo**.

Análise Semântica

* **Alocação de memória**
  * Coleta de lixo
    * Existem linguagens seguras quanto ao tipo, como ML, para as quais podemos determinar os tipos em tempo de compilação. Há outras linguagens seguras quanto ao tipo, como Java, cujos tipos não são conhecidos em tempo de compilação, mas podem ser determinados em tempo de execução.
    * Essas últimas são chamadas linguagens dinamicamente tipadas. Se uma linguagem não for estaticamente ou dinamicamente segura quanto ao tipo, então ela é considerada insegura.

Análise Semântica

* **Alocação de memória**
  * Coleta de lixo
    * Linguagens inseguras, que infelizmente incluem algumas das linguagens mais importantes como C e C++, são más candidatas à coleta de lixo automática.

    * Nas linguagens inseguras, os endereços de memória podem ser manipulados arbitrariamente: operações aritméticas arbitrárias podem ser aplicadas aos apontadores para criar novos apontadores, e inteiros arbitrários podem ser convertidos para apontadores.

Análise Semântica

* **Alocação de memória**
  * Coleta de lixo
    * Assim, um programa teoricamente poderia referir-se a qualquer endereço na memória a qualquer momento. Consequentemente, nenhum endereço da memória pode ser considerado inacessível, e nenhuma porção de memória pode sequer ser reivindicada com segurança.

    * Na prática, a maioria das linguagens em C e C++ não gera apontadores arbitrariamente, e um coletor de lixo teoricamente inseguro, que funciona bem empiricamente, pode ser desenvolvido e usado.

Análise Semântica

* **Alocação de memória**
  * Coleta de lixo
    * A coleta de lixo frequentemente é tão dispendiosa que, embora tenha sido inventada há décadas e absolutamente impeça vazamentos de memória, ainda não foi adotada por muitas das principais linguagens de programação.
    * Várias abordagens diferentes foram propostas com o passar dos anos, e não existe um algoritmo de coleta de lixo claramente melhor.

Análise Semântica

* **Alocação de memória**
  * Coleta de lixo
    * Antes de explorar as opções existentes, vamos enumerar as métricas de desempenho que devem ser consideradas quando se projeta um coletor de lixo.
      * **Tempo de execução geral**. A coleta de lixo pode ser muito lenta. É importante que ela não aumente significativamente o tempo de execução total de uma aplicação. Como o coletor de lixo necessariamente precisa manusear muitos dados, seu desempenho é determinado em grande parte pelo modo como ele aproveita o subsistema de memória.
      * **Uso de espaço**. É importante que a coleta de lixo evite a fragmentação e faça o melhor uso possível da memória disponível.

Análise Semântica

* **Alocação de memória**
  * Coleta de lixo
    * Antes de explorar as opções existentes, vamos enumerar as métricas de desempenho que devem ser consideradas quando se projeta um coletor de lixo.
      * **Tempo de pausa**. Coletores de lixo simples são notórios por fazer com que os programas parem de repente por um tempo extremamente longo, porque a coleta de lixo entra sem aviso. Assim, além de minimizar o tempo execução geral, é desejável que o tempo máximo de pausa seja minimizado. Como um caso especial importante, as aplicações de tempo real exigem que certas computações sejam completadas dentro de um limite de tempo. Temos de suprimir a coleta de lixo enquanto realizamos tarefas em tempo real ou restringir ao máximo o tempo de pausa. Assim, a coleta de lixo raramente é usada em aplicações de tempo real.

Análise Semântica

* **Alocação de memória**
  * Coleta de lixo
    * Antes de explorar as opções existentes, vamos enumerar as métricas de desempenho que devem ser consideradas quando se projeta um coletor de lixo.
      * **Localidade do programa**. Não podemos avaliar a velocidade de um coletor de lixo unicamente por seu tempo de execução. O coletor de lixo controla o posicionamento de dados e, assim, influencia a localidade de dados do programa. Ele pode melhorar a localidade temporal de um programa, liberando espaço e reutilizando-o, assim como pode melhorar a localidade espacial do programa, relocando dados usados juntos, na mesma cache ou página.

Análise Semântica

* **Alocação de memória**
  * Coleta de lixo
    * Alguns desses objetivos de projeto entram em conflito entre si, e as escolhas precisam ser feitas com cuidado, considerando-se como os programas tipicamente se comportam.
    * Além disso, objetos com características diferentes podem favorecer tratamentos diferentes, exigindo que um coletor use diferentes técnicas para diferentes tipos de objetos.

Análise Semântica

* **Alocação de memória**
  * Coleta de lixo
    * Por exemplo, o número de objetos alocados é dominado por pequenos objetos, de forma que a alocação de objetos pequenos não deverá ocasionar um grande custo.
    * Por outro lado, considere os coletores de lixo que relocam objetos alcançáveis. A relocação é cara quando se trata de objetos grandes, porém menos custosa para objetos pequenos.

Análise Semântica

* **Alocação de memória**
  * Coleta de lixo
    * Como outro exemplo, em geral quanto mais tempo se espera para coletar lixo em um coletor baseado em rastreamento, maior é a fração de objetos que podem ser coletados.
    * O motivo é que os objetos frequentemente 'morrem cedo': assim, se esperarmos um tempo, muitos dos objetos recém-alocados se tornarão inalcançáveis.

Análise Semântica

* **Alocação de memória**
  * Coleta de lixo
    * Tal coletor custa menos, na média, por objeto inalcançável coletado. Por outro lado, a coleta infrequente aumenta o uso de memória de um programa, diminui sua localidade de dados e aumenta a extensão das pausas.

    * Ao contrário, um coletor de contagem de referência, introduzindo um custo constante a muitas das operações do programa, pode atrasar significativamente a execução geral de um programa.

Análise Semântica

* **Alocação de memória**
  * Coleta de lixo
    * Por outro lado, a contagem de referência não cria longas pausas, e sua memória é eficiente, porque encontra lixo assim que ele é produzido (com exceção de certas estruturas cíclicas).

    * **O projeto da linguagem também pode afetas as características do uso de memória**. Algumas linguagens encorajam um estilo de programação que gera muito lixo.

Análise Semântica

* **Alocação de memória**
  * Coleta de lixo
    * Por exemplo, os programas em linguagens funcionais (ou quase funcionais) criam mais objetos para evitar a mutação dos objetos existentes.
    * Em Java, todos os objetos, que não os tipos básicos como inteiros e referências, são alocados no heap e não na pilha, mesmo que seus tempos de vida sejam confinados aos de uma chamada de função.

Análise Semântica

* **Alocação de memória**
  * Coleta de lixo
    * Projetos dessa natureza liberam o programador da preocupação com os tempos de vida das variáveis à custa de mais lixo.
    * Algumas técnicas de otimização de compilador foram desenvolvidas para analisar os tempos de vida das variáveis e alocá-las na pilha sempre que possível.

Análise Semântica

* **Análise semântica em tipos de variáveis**
  * Para fazer a verificação de tipo, um compilador precisa atribuir uma expressão de tipo a cada componente do programa fonte.
  * O compilador precisa, então, determinar se essas expressões estão em conformidade com uma coleção de regras lógicas, que chamamos de sistema de tipo, para a linguagem fonte.

Análise Semântica

* **Análise semântica em tipos de variáveis**
  * A verificação de tipo tem o potencial de detectar erros de tipos nos programas. A princípio, qualquer verificação pode ser feita dinamicamente, se o código objeto mantiver o tipo de um elemento juntamente com o seu valor.

  * Um sistema de tipo seguro elimina a necessidade da verificação dinâmica para erros de tipo, porque nos permite detectar estaticamente que esses erros não podem ocorrer quando o programa objeto é executado.

Análise Semântica

* **Análise semântica em tipos de variáveis**
  * Uma implementação de uma linguagem de programação é fortemente tipada (strongly typed) se um compilador garantir que os programas que ela aceita executarão sem erros de tipo.
  * Além de sua utilidade na compilação, ideias sobre a verificação de tipo têm sido usadas para melhorar a segurança de sistemas que permitem que módulos de software sejam importados e executados.

Análise Semântica

* **Análise semântica em tipos de variáveis**
  * Os programas Java são compilados para byte-codes independentes de máquina, os quais incluem informações detalhadas dos tipos relativos às operações.
  * O código importado é verificado antes de sua permissão para executar, para proteger contra erros inadvertidos e comportamento malicioso.

Análise Semântica

* **Análise semântica em tipos de variáveis**
  * A verificação de tipo pode assumir duas formas: síntese e inferência. A síntese de tipo constrói o tipo de uma expressão a partir dos tipos de suas subexpressões. Ela exige que os nomes sejam declarados antes de serem usados.

  * O tipo de *E*1 + *E*2 é definido em termos dos tipos de *E*1 e *E*2. Uma regra típica para a síntese de tipo tem a forma:

if f tem tipo $s \rightarrow t$ **and** x tem tipo s,
then expressão $f(x)$ tem tipo t

Análise Semântica

* **Análise semântica em tipos de variáveis**
  * Nesta regra, *f* e *x* denotam expressões, e *s* -> *t* denota uma função de *s* para *t*. Essa regra para funções com um argumento também é válida para funções com vários argumentos.

  * A regra apresentada anteriormente pode ser adaptada para *E*1 + *E*2 visualizando-a como uma aplicação da função add(*E*1, *E*2).

Análise Semântica

* **Análise semântica em tipos de variáveis**
  * A inferência de tipo determina o tipo de uma construção da linguagem a partir do modo como ela é usada.

  * As variáveis representando expressões de tipo nos permitem falar a respeito de tipos desconhecidos. Usaremos letras gregas α e β para variáveis de tipo nas expressões de tipo. Uma regra típica para inferência de tipo tem a forma:

if $f(x)$ é uma expressão,
then para algum α e β, f tem tipo $\alpha \rightarrow \beta$ **and** x tem tipo α

Análise Semântica

* **Análise semântica em tipos de variáveis**
  * A inferência de tipo é necessária para linguagens como ML, que verificam tipos, mas não exigem que nomes sejam declarados.

  * A seguir, consideraremos a verificação de tipo nas expressões. As regras para verificar os comandos são semelhantes àquelas para expressões. Por exemplo, o comando condicional: '**if** (*E*) *S*;' é tratado como se ele fosse a aplicação de uma função *if* a *E* e *S*. O tipo especial void indica a ausência de um valor. Então, a função if espera ser aplicada a um booliano e a um void; o resultado da aplicação é um void.

Análise Semântica

* **Análise semântica em tipos de variáveis**
  * Considere as expressões como *x* + *i*, onde *x* é do tipo float e *i* é do tipo integer.

  * Como a representação dos números inteiros e de ponto flutuante é diferente internamente em um computador, e diferentes instruções de máquina são usadas para operações com inteiros e ponto flutuante, o compilador pode ter de converter um dos operandos de + para garantir que os dois operandos sejam do mesmo tipo quando ocorrer a adição.

Análise Semântica

* **Análise semântica em tipos de variáveis**
  * Suponha que os inteiros sejam convertidos para pontos flutuantes quando necessário, usando um operador unário (float). Por exemplo, o inteiro 2 é convertido para um ponto flutuante no código para a expressão 2 + 3.14:

```
t1 = (float) 2
t2 = t1 * 3.14
```

  * Podemos estender esses exemplos para considerar as versões de tipos inteiro e ponto flutuante dos operadores; por exemplo, int para operandos inteiros e float para ponto flutuante.

Análise Semântica

* **Análise semântica em tipos de variáveis**
  * Vejamos a síntese de tipos para traduzir expressões.

  * Introduziremos outro atributo *E*.type, cujo valor é integer ou float. A regra associada a *E* -> *E*1 + *E*2 é construída com o pseudocódigo:

if ($E_1.type = integer$ **and** $E_2.type = integer$) $E.type = integer$;
else if ($E_1.type = float$ **and** $E_2.type = integer$) ...
...

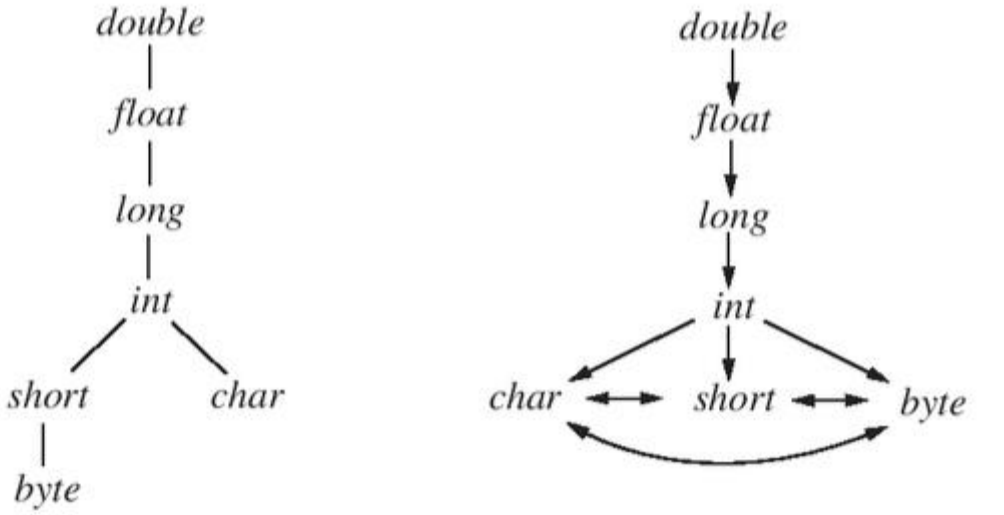
Análise Semântica

* **Análise semântica em tipos de variáveis**
  * À medida que o número de tipos sujeitos a conversão aumenta, o número de casos aumenta rapidamente. Portanto, com grandes quantidades de tipos, uma organização cuidadosa das ações semânticas se torna importante.

  * As regras de conversão de tipo variam de uma linguagem para outra. As regras para Java na Fig. 2 fazem a distinção entre conversões de alargamento que visam preservar informações, e conversões de estreitamento, que podem perder informações. As regras de alargamento são dadas pela hierarquia na Fig. 2(a) a seguir:

Análise Semântica

* **Análise semântica em tipos de variáveis**
  * Abaixo, Fig.2:



(a) Conversões de alargamento (b) Conversões de estreitamento

Conversões entre os tipos primitivos de Java.

Análise Semântica

* **Análise semântica em tipos de variáveis**
  * Qualquer tipo mais abaixo na hierarquia pode ser alargado para um tipo mais acima na mesma hierarquia. Assim, um char pode ser alargado para um int ou para um float, mas um char não pode ser alargado para um short.

  * As regras de estreitamento são ilustradas pelo grafo na Fig. 2(b): um tipo *s* pode ser estreitado para um tipo *t* se existe um caminho de *s* para *t*. Observe que char, short, e byte são conversíveis entre si.

Análise Semântica

* **Análise semântica em tipos de variáveis**
  * A conversão de um tipo para outro é considerada **implícita** se for feita automaticamente pelo compilador. As conversões de tipo implícitas, também chamadas coerções, são limitadas em muitas linguagens a conversões de alargamento.

  * A conversão é considerada **explícita** se o programador tiver de escrever algum fragmento de código para causar a conversão. As conversões explícitas também são chamadas *casts*.

Análise Semântica

* **Análise semântica em comandos**
  * A análise semântica também deve ser aplicada aos **comandos** da linguagem.
  * O compilador deve garantir que um comando **break**, por exemplo, esteja incorporado em um comando **while**, **for** ou **switch**. Se não houver uma dessas instruções envolventes, um erro é informado.

Análise Semântica

* **Análise semântica em comandos**
  * Ações semânticas devem ser aplicadas também à sobrecarga de funções e operadores.
  * Um símbolo sobrecarregado possui **diferentes significados**, dependendo do seu contexto. A sobrecarga é resolvida quando um único significado é determinado para cada ocorrência de um nome.
  * A sobrecarga de métodos deve gerar novas entradas na tabela de símbolos.

Análise Semântica

* **Análise semântica em comandos**
  * Vejamos dois exemplos:
    * O operador + em Java denota concatenação de cadeia ou adição, dependendo dos tipos de seus operandos.
    * As funções definidas pelo programador também podem ser sobrecarregadas, como em:

```
void err() {... }
void err(String s) {... }
```

    * Observe que podemos escolher entre essas duas versões da função **err** examinando seus argumentos.

Análise Semântica

* **Análise semântica em comandos**
  * A assinatura para uma função consiste no nome da função e nos tipos de seus argumentos. A suposição de que podemos resolver a sobrecarga baseada nos tipos dos argumentos é equivalente a dizer que podemos resolver a sobrecarga baseada nas assinaturas.

  * Nem sempre é possível resolver a sobrecarga examinando apenas os argumentos de uma função. Na linguagem de programação Ada, em vez de um tipo único uma subexpressão isolada pode ter um conjunto de tipos possíveis para os quais o contexto precisa fornecer informações suficientes para reduzir as opções a um único tipo.

Análise Semântica

* **Análise semântica em comandos**
  * Outras importantes ações semânticas dizem respeito à inferência de tipo e funções polimórficas.
  * A inferência de tipo é útil para linguagens fortemente tipadas que não exigem que os nomes sejam declarados antes de serem usados. A inferência de tipo garante que os nomes sejam usados de forma coerente.
  * O termo 'polimórfico' refere-se a qualquer fragmento de código que pode ser executado com argumentos de diferentes tipos.

Análise Semântica

* **Principais erros semânticos**
  * **Escopo dos identificadores**
    * O compilador deve garantir que variáveis e funções estejam declaradas em locais que podem ser acessados onde esses identificadores estão sendo utilizados. Exemplo: Uma classe com funções declaradas como privadas e essas funções sendo utilizadas fora da classe. Ou, de forma análoga, um atributo declarado como privado sendo utilizado fora do seu escopo.

Análise Semântica

* **Principais erros semânticos**
  * **Compatibilidade de tipos**
    * Verificar se os tipos de dados declarados nas variáveis e funções estão sendo utilizados e atribuídos corretamente, por exemplo: operações matemáticas devem ser realizadas com números. Ou, de forma análoga, variável declarada como int recebendo um valor string.

Análise Semântica

* **Principais erros semânticos**
  * **Compatibilidade de tipos**

```
var i : int;
var s : string;
s = “joao";
i := s;
```

  * No código acima qual é o comportamento em uma linguagem interpretada como Python? E no Pascal que é compilado?

Análise Semântica

* **Principais erros semânticos**
  * **Compatibilidade de tipos**
    * Em alguns casos pode ser efetuada a conversão de tipos, essa operação é conhecida como *cast*, e pode ser feita de forma explícita no código ou pelo próprio compilador.

    * Atualmente, a incompatibilidade de tipos é um dos fatores que mais causam abertura de bugs em aplicações grandes. A conversão de tipos deve ser feita com parcimônia pelo programador.

Análise Semântica

* **Principais erros semânticos**
  * **Detectar unicidade de nomes de identificadores**
    * Essa verificação é muito importante pois ele deve garantir que os identificadores sejam únicos não havendo na tabela de símbolos uma entrada para o mesmo identificador.
  * **Detectar o uso correto de comandos de controle de fluxo**
    * Comandos como **continue** e **break** executam saltos na execução do código, esses comandos devem ser utilizados em instruções que permitam estes saltos.

Resumo

* ***Organização em tempo de execução***: Para implementar as abstrações incorporadas na linguagem fonte, um compilador cria e gerencia um ambiente em tempo de execução em cooperação com o sistema operacional e a máquina alvo. O ambiente em tempo de execução possui áreas de dados estáticas para o código objeto e objetos de dados estáticos criadas em tempo de compilação. Também possui áreas dinâmicas de pilha e de heap para gerenciar objetos criados e destruídos enquanto o programa objeto é executado.

* ***Pilha de controle***: Chamadas e retornos de procedimento normalmente são gerenciados por uma pilha em tempo de execução chamada pilha de controle. Podemos usar uma pilha porque as chamadas de procedimento ou ativações são aninhadas no tempo, ou seja, se p chama q, então essa ativação de q é aninhada dentro dessa ativação de p.

Resumo

* ***Alocação de pilha***: A memória para variáveis locais pode ser alocada em uma pilha de execução para linguagens que permitam ou exijam que as variáveis locais se tornem inacessíveis quando seus procedimentos terminarem. Para essas linguagens, cada ativação viva possui um registro de ativação (ou frame) na pilha de controle, com a raiz da árvore de ativação no fundo da pilha, e toda a sequência de registros de ativação correspondendo ao caminho na árvore de ativação até a ativação onde o controle corretamente reside. A última ativação tem seu registro no topo da pilha.

* ***Gerenciamento da heap***: O heap é uma porção de memória usada para os dados que podem estar vivos indefinidamente, ou até que o programa os exclua explicitamente. O gerenciador de memória aloca e libera espaço dentro do heap. A coleta de lixo encontra espaços dentro do heap que não estão mais em uso e, portanto, podem ser liberados para acomodar outros itens de dados. Para linguagens que exigem isso, o coletor de lixo é um importante subsistema do gerenciador de memória.

Resumo

* ***Explorando a localidade***: Fazendo bom uso da hierarquia de memória, os gerenciadores de memória podem influenciar no tempo de execução de um programa. O tempo gasto para acessar diferentes partes da memória pode variar de nanossegundos a milissegundos. Felizmente, a maioria dos programas gasta a maior parte de seu tempo executando uma fração relativamente pequena do código e usando apenas uma pequena fração dos dados. Um programa possui localidade temporal se for provável que ele acesse os mesmos endereços de memória novamente em breve. Um programa possui localidade espacial se for provável que ele acesse endereços vizinhos de memória em breve.

* ***Reduzindo a fragmentação***: À medida que o programa aloca e libera memória, o heap pode tornar-se fragmentado ou quebrado em grandes quantidades de pequenos espaços livres não contíguos, ou buracos. A estratégia de alocar o menor buraco disponível que satisfaça uma solicitação tem funcionado bem empiricamente. Porém, embora costume melhorar a utilização de espaço, pode não ser a melhor para a localidade espacial. A fragmentação pode ser reduzida combinando-se ou unindo-se buracos adjacentes.

Resumo

* ***Liberação manual***: O gerenciamento manual da memória tem duas falhas comuns: não remover dados que não podem ser referenciados, o que é chamado erro de vazamento de memória, e referenciar dados removidos, o que caracteriza um erro de acesso a apontador pendente.

* ***Alcançabilidade***: Lixo é o nome dado ao que não pode ser referenciado ou alcançado. Existem duas maneiras básicas de encontrar objetos inalcançáveis: capturar a transição quando um objeto alcançável se tornar inalcançável, ou localizar periodicamente todos os objetos alcançáveis e inferir que todos os objetos restantes são inalcançáveis.

* ***Coletores por contagem de referência***: Essa classe de coletores mantêm um contador das referências a um objeto; quando o contador passa para zero, o objeto se torna inalcançável. Esses coletores introduzem o custo de administrar referências e podem deixar de encontrar lixo 'cíclico', o qual consiste em objetos inalcançáveis que referenciam uns aos outros, talvez por meio de uma cadeia de referências.

Resumo

* ***Coletores de lixo baseados em rastreamento***: Estes examinam ou rastreiam iterativamente todas as referências para encontrar objetos alcançáveis, começando no conjunto raiz consistindo em objetos que podem ser acessados diretamente sem a necessidade de seguir apontadores.

* ***Estático x Dinâmico***: O sistema de tipos de dados podem ser divididos em dois grupos: sistemas estáticos e sistemas dinâmicos. Muitas das linguagens utilizam o sistema estático. Esse sistema é predominante em linguagens compiladas, pois essa informação é utilizada durante a compilação e simplifica o trabalho do compilador. Sistema de tipo estático: Linguagem como C, Java, Pascal obrigam o programador a definir os tipos das variáveis e retorno de funções, o compilador pode fazer várias checagens de tipo em tempo de compilação. Sistema de tipo dinâmico: Variáveis e retorno de funções não possuem declaração de tipos, como exemplo temos linguagens como Python , Perl e PHP.

Resumo

* ***Inferência de tipos***: Algumas linguagens utilizam um mecanismo muito interessantes chamado inferência de tipos, que permite a uma variável assumir vários tipos durante o seu ciclo de vida, isso permite que a ela possa ter vários valores. Linguagens de programação como Haskel tira proveito desse mecanismo. Nesses casos o compilador infere o tipo da variável em tempo de execução, esse tipo de mecanismo está diretamente relacionado ao mecanismo de Generics do Java e Delphi Language. A validação de tipos passa a ser realizada em tempo de execução.

* ***Tabela de Símbolos e Análise Semântica***: A análise semântica, assim como as demais fases, necessita coletar informações presentes na tabela de símbolos. Na tabela de símbolos são guardados os nomes dos objetos definidos pelo programador e é através dela que eles são referenciados simbolicamente, por nome, ao longo de todo o programa. Nos casos mais simples, as tabelas de símbolos são meros vetores linearmente organizados a partir de elementos de comprimento fixo, com política de acesso aleatório para leitura. Há compiladores que utilizam, para armazenar as tabelas, uma estrutura de dados com política de acesso de pilha, que apresenta a vantagem de se tornar disponível para uso em outras aplicações simultâneas.

Obrigado.

joaopauloaramuni@gmail.com
joaopauloaramuni@fumec.br

UNIVERSIDADE FUMEC
50 Anos