Aula 00 – Introdução

Prof. Eduardo Zambon

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (Ufes)

Compiladores
***Compiler Construction* (CC)**

Introdução

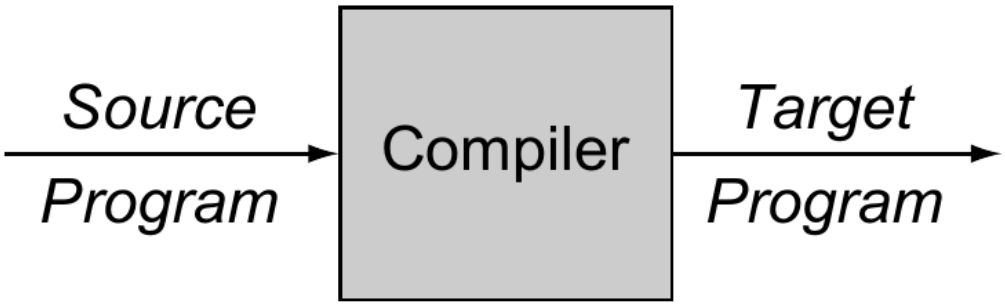
- Compiladores são um componente fundamental de qualquer sistema de computação.
- **Estes *slides***: introdução sobre o funcionamento e organização geral de um compilador.
- **Objetivos**: fixar termos, definições e notações para o restante do curso.

Referências

Chapter 1 – Introduction
K. C. Louden
Chapter 1 – Overview of Compilation
K. D. Cooper
Chapters 1 & 2
D. Thain

Introdução

- Compiladores são programas que traduzem uma linguagem em outra.
- Recebem como entrada um programa escrito na linguagem fonte (*source language*).
- Produzem como saída um programa **equivalente** escrito na linguagem alvo (*target language*).
- Em geral, a linguagem fonte é uma linguagem de alto-nível: C, C++, etc.
- Linguagem alvo é código *assembly*, i.e., código de máquina.
- Até agora, o compilador era visto como uma caixa-preta.



Introdução

- Três pilares da programação de sistemas: hardware, compilador e sistema operacional.
- Compiladores são programas extremamente complexos.
- $\Rightarrow$ Construção de um compilador deve ser devidamente projetada.
- Projeto também facilita o entendimento do compilador.
- Compiladores podem ter milhares e milhares de linhas de código.
- *Exemplo*: GCC 4.9 – 14.5 milhões (!) de linhas de código.
- *Em contraste*: Linux 4.7 – 21.7 milhões de linhas.

Introdução

Por que estudar compiladores?

- Parte da formação fundamental dos cursos de computação.
- Construção de um compilador engloba/integra várias disciplinas do curso.
  - Linguagens Formais e Autômatos
  - Sistemas Operacionais
  - Estrutura de Dados
  - Arquitetura de Computadores
  - Etc, etc, . . .
- Algoritmos/métodos/ferramentas estudados podem ser aplicados separadamente em variadas situações.

Introdução

Como estudar compiladores?

- Aspectos teóricos:
  - Organização de compiladores.
  - Teoria de linguagens e reconhecedores.
  - Algoritmos clássicos utilizados.
- Aspectos práticos:
  - Exemplos de compiladores reais.
  - Exercícios práticos (laboratório): construção de partes de um compilador.
  - Trabalho prático: implementação de um compilador completo.

Breve História dos Compiladores

- Originalmente os programas eram escritos em linguagem de máquina (*machine language*).
- *Exemplo* (em Intel 8x86):

```
C7 06 0000 0002
```

- Comando acima é uma instrução para mover (`C7 06`) o valor `0002` para o endereço `0000`.
- A seguir, vieram as linguagens de montagem (*assembly languages*).
- *Exemplo* (assumindo que a variável `X` corresponde ao endereço `0000`):

```
MOV X, 2
```

- Tradução de *assembly* para linguagem de máquina é feita pelo montador (*assembler*).

Breve História dos Compiladores

- Programar em linguagem *assembly* não é trivial.
- Pior problema: portabilidade.
- Levou ao surgimento das linguagens de alto nível (*high-level languages*).
- *Exemplo*:

```
X = 2
```

- O primeiro compilador foi desenvolvido entre 1954 e 1957.
- Linguagem FORTRAN e seu compilador: John Backus e time na IBM.
- Por volta da mesma época: estudo da estrutura de linguagem natural por Noam Chomsky.

Breve História dos Compiladores

- Teorias e algoritmos relacionados ao processamento de linguagens surgiram nas décadas de 1960 e 1970.
- Classificação de linguagens: Hierarquia de Chomsky (HC).
- *Scanning* – métodos simbólicos para expressar a estrutura das palavras de uma linguagem de programação (LP):
  - Linguagens regulares (tipo 3 na HC).
  - Representadas por expressões regulares.
  - Reconhecidas por autômatos finitos.
- *Parsing* – mecanismos para reconhecimento da estrutura sintática da LP:
  - Linguagens livres de contexto (tipo 2 na HC).
  - Representadas por gramáticas livres de contexto.
  - Reconhecidas por autômatos de pilha (*pushdown automata*).

Breve História dos Compiladores

- Programas foram desenvolvidos para automatizar parte do desenvolvimento do compilador.
- Geradores de *scanners*, tal como o `lex` (atualmente `flex`) desenvolvido por Mike Lesk para o Unix em 1975.
- Geradores de *parsers*, tal como o `yacc` (atualmente `bison`) desenvolvido por Steve Johnson também para Unix por volta da mesma época.
- Demais projetos focaram em automatizar a geração de outras partes do compilador.
- Fim da década de 1970, início da década de 1980: tentativa de automatizar a geração de código executável.
- Bem menos sucesso: problema menos entendido.

Breve História dos Compiladores

Avanços mais recentes no projeto de compiladores:

- Algoritmos mais sofisticados para inferência e/ou simplificação da informação contida em um programa.
- *Exemplo*: algoritmo de Hindley-Milner para unificação de verificação de tipos. Muito usado em linguagens funcionais.
- Ambientes Integrados de Desenvolvimento (IDEs): incluem editores, compiladores, gerenciadores de pacotes, etc.
- $\Rightarrow$ Projeto básico de um compilador não mudou muito nos últimos 20 anos.

Programas Relacionados a Compiladores

Interpretadores:

- Executam o programa fonte imediatamente ao invés de gerar código binário.

- *Exemplo*: BASIC.
- Interpretação pura tem um tempo de execução bem mais lento. Em geral por um fator de 10 ou mais.
- $\Rightarrow$ Uso de sistemas híbridos: Java, Python, etc.
- Interpretadores realizam praticamente as mesmas operações que os compiladores.

Programas Relacionados a Compiladores

Montadores (*Assemblers*):

- Um tradutor para a linguagem *assembly* de uma arquitetura em particular.
- Linguagem *assembly* é uma forma simbólica de uma linguagem de máquina.
- Um compilador pode gerar linguagem *assembly* como a sua linguagem alvo e um *assembler* termina a tradução em código de máquina.
- O montador da suite GCC é o `as`.

Programas Relacionados a Compiladores

Ligadores (*Linkers*):

- Compõem arquivos objetos separados em um arquivo diretamente executável.
- Conectam um programa objeto com o código de funções padrões de bibliotecas e com recursos providos pelo SO.
- Corresponde à última etapa na geração do código executável.
- Totalmente dependente da plataforma: SO + arquitetura.
- No GCC: `ld`.

Programas Relacionados a Compiladores

Carregadores (*Loaders*):

- Resolvem todos os endereços realocáveis relativos a um dado endereço base.
- Tornam o código executável mais flexível.
- Geralmente são parte do sistema operacional e não um programa separado.

Pré-processadores:

- Apagam comentários, incluem outros arquivos e realizam substituições de macros.
- Requeridos por algumas linguagens (como em C) ou podem ser adicionados depois para prover funcionalidades adicionais.

Outras ferramentas: editores, depuradores (*debuggers*), *profilers*, gerenciadores de pacotes, controle de versão, etc.

Na Prática

Considere o seguinte arquivo `hello.c`:

```
#include <stdio.h>
int main(void) {
    printf("Hello, World!\n");
}
```

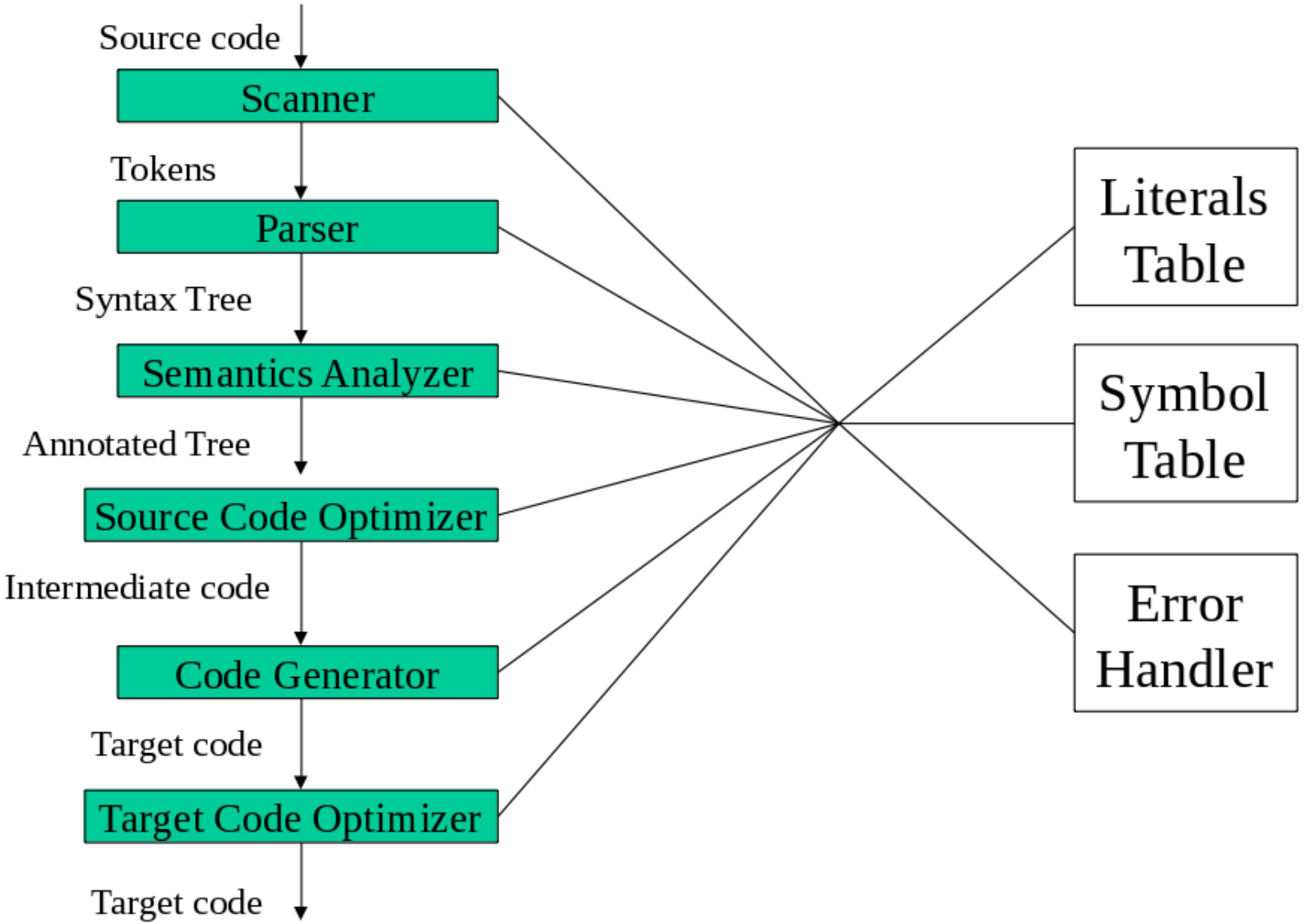
Execute os comandos abaixo e veja o resultado:

- **`gcc -E hello.c > hello.pre.c`**: executa somente o pré-processador.
- **`gcc -S hello.c`**: compila mas não chama o *assembler* nem o *linker*.
- **`gcc -c hello.c`**: compila e chama o *assembler* mas não o *linker*. (`objdump -d hello.o` para inverso.)
- **`gcc hello.c`**: gera o executável dinâmico.
- **`gcc --static hello.c`**: gera o executável estático.

As Fases de um Compilador

- O processo de compilação pode (deve!) ser dividido em diferentes etapas/fases.
- Essa divisão permite uma construção modular do compilador.
- Em geral, um compilador pode ser dividido em seis módulos:
  - *Scanner* (analisador léxico)
  - *Parser* (analisador sintático)
  - Analisador semântico
  - Otimizador de código fonte
  - Gerador de código
  - Otimizador de código alvo
- Nem todos os módulos acima precisam necessariamente estar presentes.
- Componentes auxiliares de um compilador: tabela de literais (*strings*), tabela de símbolos, tratador de erros.

As Fases de um Compilador



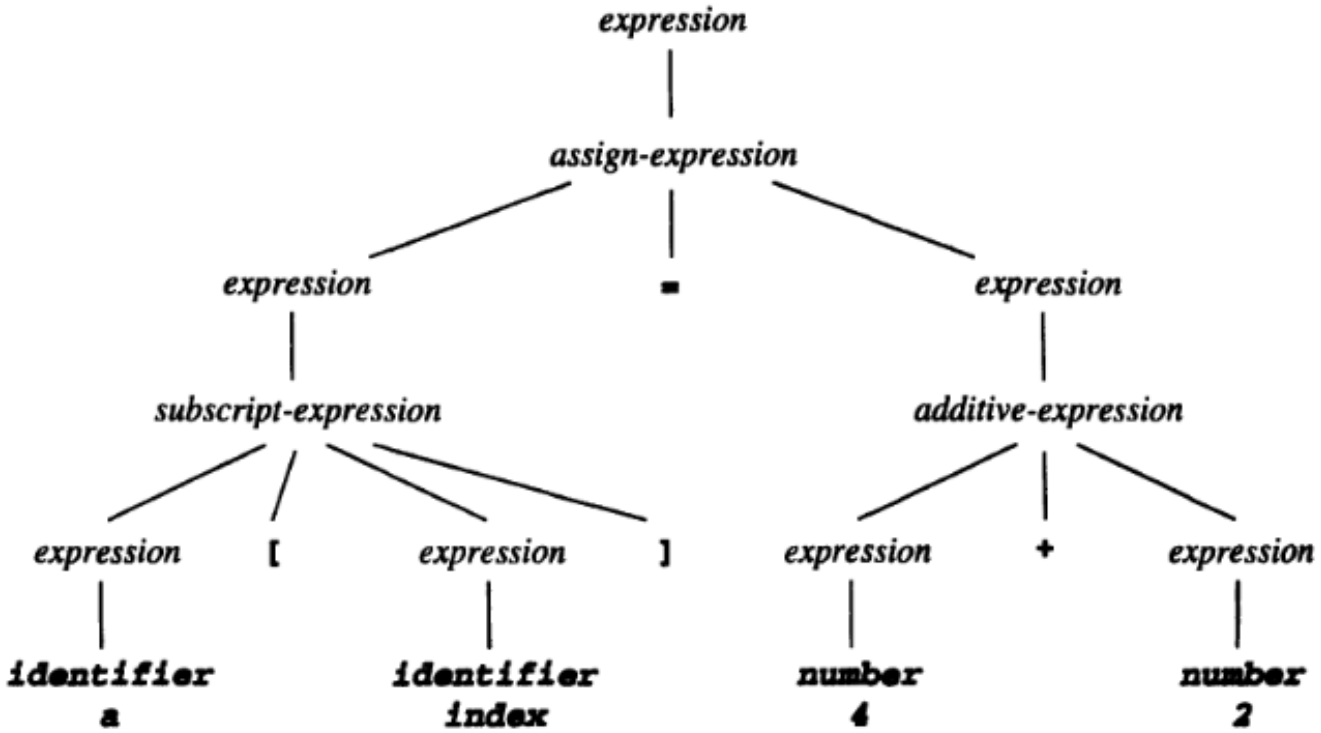
Scanner – Analisador Léxico

- Análise léxica: reúne sequências de caracteres em unidades significativas chamadas *tokens*.
- *Scanner* também pode inserir *strings* na tabela de literais.
- Cada *token* é formado por:
  - Lexema: *string* que representa o *token*.
  - Categoria sintática: também chamado de tipo do *token*.
- *Exemplo*: o comando `a[index] = 4 + 2` pode ser dividido em *tokens* como abaixo.

Lexema	**Tipo**
`a`	identificador
`[`	colchete à esquerda
`index`	identificador
`]`	colchete à direita
`=`	atribuição
`4`	número
`+`	sinal de adição
`2`	número

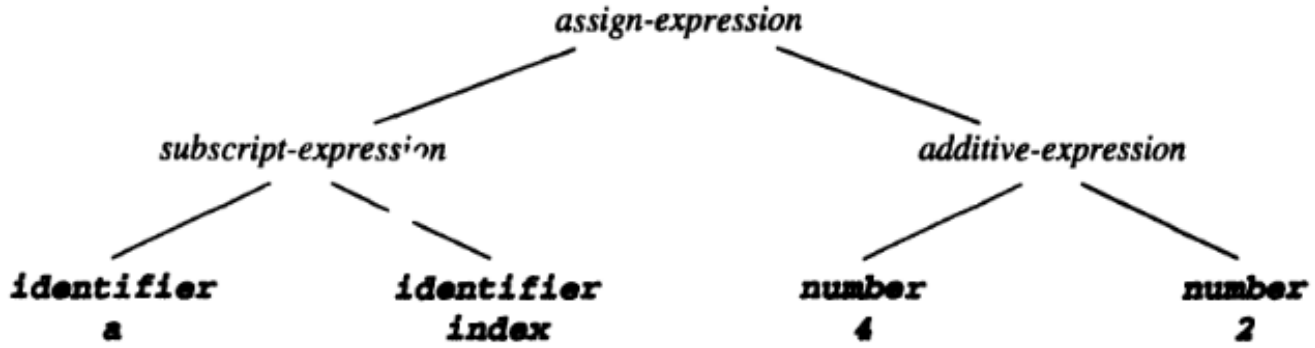
Parser – Analisador Sintático

- Análise sintática: determina a estrutura do programa.
- Possíveis resultados da análise sintática:
  - Árvore de derivação (*parse tree*).
  - Árvore de sintaxe abstrata (*abstract syntax tree – AST*).
- *Exemplo*: *parse tree* do comando `a[index] = 4 + 2`:



Parser – Analisador Sintático

- Nós folha da *parse tree*: sequência de tokens do programa.
- *Parse tree* pode ser útil para visualização do programa mas possui informações desnecessárias para a compilação.
- Simplificação da *parse tree* leva à AST.
- AST é uma forma de representação intermediária muito usada.
- *Exemplo*: AST do comando `a[index] = 4 + 2`:

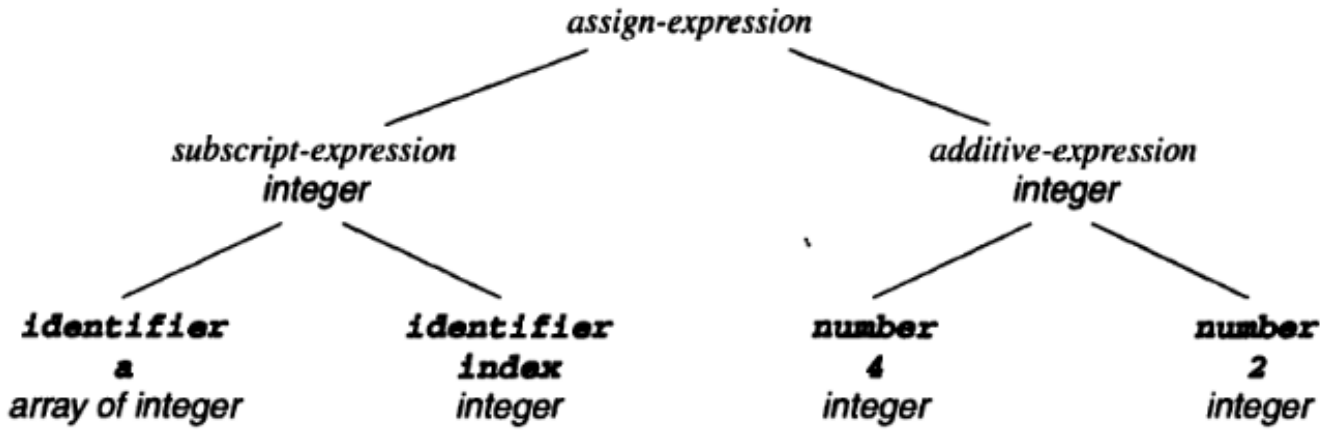


Analisador Semântico

- Sintaxe de um programa: estrutura do código.
- Semântica de um programa: significado do código.
- Semântica determina o comportamento do programa durante a execução.
- Análise estática (*static analysis*): elementos da LP que não podem ser expressos como sintaxe mas que podem ser analisados antes da execução.
- Elementos típicos de análise estática: verificação de tipos e declaração de variáveis.
- Análise dinâmica: verificações que podem ser feitas somente em tempo de execução (*run-time*).
- *Exemplo*: comando **`instanceof`** no Java.

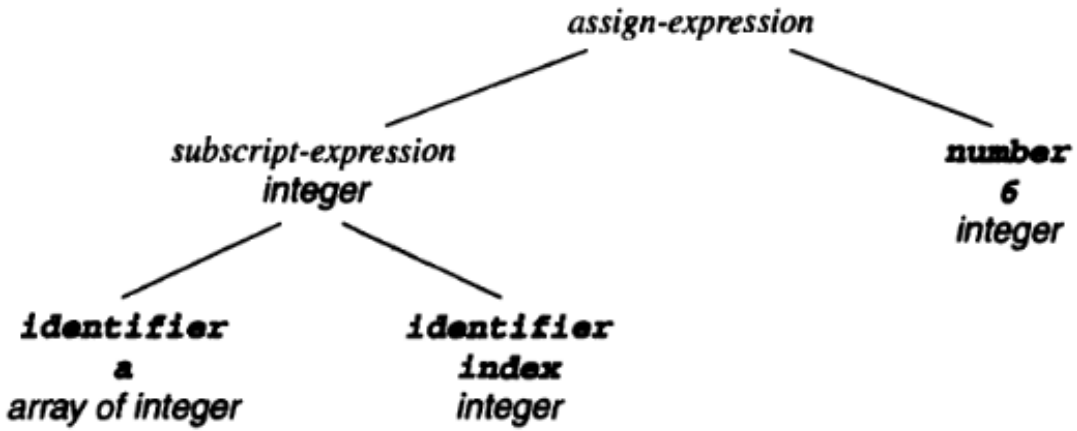
Analisador Semântico

- Atributos: informações adicionais computadas pelos analisador semântico.
- Geralmente são adicionados à AST.
- AST + atributos = *Annotated Tree*.
- *Exemplo*: *Annotated tree* de `a[index] = 4 + 2`:



Otimizador de Código Fonte

- Após análise semântica: programa correto do ponto de vista da especificação da LP.
- Programa correto $\Rightarrow$ possibilidade de otimizações.
- Melhorias do código que dependem apenas do código fonte podem ser feitas nesta etapa.
- Diferentes compiladores exibem uma grande variação nos tipos de otimizações realizadas.
- *Exemplo*: *Constant folding* na *annotated tree*:



Otimizador de Código Fonte

- Algumas otimizações podem ser feitas diretamente na árvore.
- Em outros casos, é mais fácil otimizar se o programa seguir uma estrutura mais linear.
- Código intermediário: representação do código próximo da linguagem de máquina.
- Código de três endereços: um tipo de código intermediário.
- *Exemplo*: *Constant folding* no código de três endereços:

```
t = 4 + 2
a[index] = t
```

```
t = 6
a[index] = t
```

```
a[index] = 6
```

Gerador de Código

- Recebe como entrada a representação intermediária (código de três endereços ou *annotated tree*) e produz código para a máquina alvo.
- Nessa fase as propriedades da máquina alvo se tornam o fator principal.
- Necessidade de adequação às instruções e representações dos dados disponíveis na máquina.
- *Exemplo*: é preciso decidir como armazenar vetores.

```
MOV  R0, index   ; valor de index -> R0
MUL  R0, 2       ; dobra valor em R0
MOV  R1, &a      ; endereço de a -> R1
ADD  R1, R0      ; adiciona R0 a R1
MOV *R1, 6       ; 6 -> endereço em R1
```

- Arquitetura do exemplo acima: inteiro = 2 bytes; endereçamento byte a byte.

Otimizador de Código Alvo

- Tenta melhorar a eficiência do código alvo gerado.
- Eficiência pode ser medida de diferentes formas:
  - Diminuir o número de instruções.
  - Substituições por operações mais rápidas.
  - Diminuir o uso de registradores.
  - Diminuir o uso da memória.
  - Diminuir o consumo de energia.
  - Etc, etc, etc.
- Geralmente envolve substituições de instruções.
- *Exemplo*: trocar multiplicação por *shift* e usar outro modo de endereçamento.

```
MOV   R0, index    ; valor de index -> R0
SHL   R0           ; dobra valor em R0
MOV   &a[R0], 6    ; 6 -> endereço a + R0
```

Principais Estruturas de Dados em um Compilador

- *Tokens*:
  - Tipos de *tokens* são definidos como um valor de um tipo de dado enumerado (`enum`).
  - Armazenados como uma única variável global ou um vetor de *tokens*.
- AST:
  - Implementada como uma estrutura padrão de ponteiros.
  - Cada nó pode armazenar diferentes atributos, dependendo do tipo do nó.
- Tabela de Símbolos:
  - Armazena informações associadas com identificadores: funções, variáveis, constantes e tipos de dados.
  - Utilizada em praticamente todas as fases de compilação.
  - Operação de consulta deve ser $O(1)$.
  - Geralmente implementada como uma ou mais tabelas *hash*.

Principais Estruturas de Dados em um Compilador

- Tabela de Literais (*Strings*):
  - Armazena constantes e *strings*, reduzindo o tamanho do programa.
  - Requer operações rápidas de inserção e consulta.
- Código Intermediário:
  - Armazenado como um vetor ou uma lista encadeada de tuplas.
  - Estrutura deve tornar fácil a reordenação de instruções.
- Código de Máquina:
  - Idem ao código intermediário.

Todas essas estruturas serão estudadas em detalhes ao longo das próximas aulas.

Análise e Síntese

Análise:

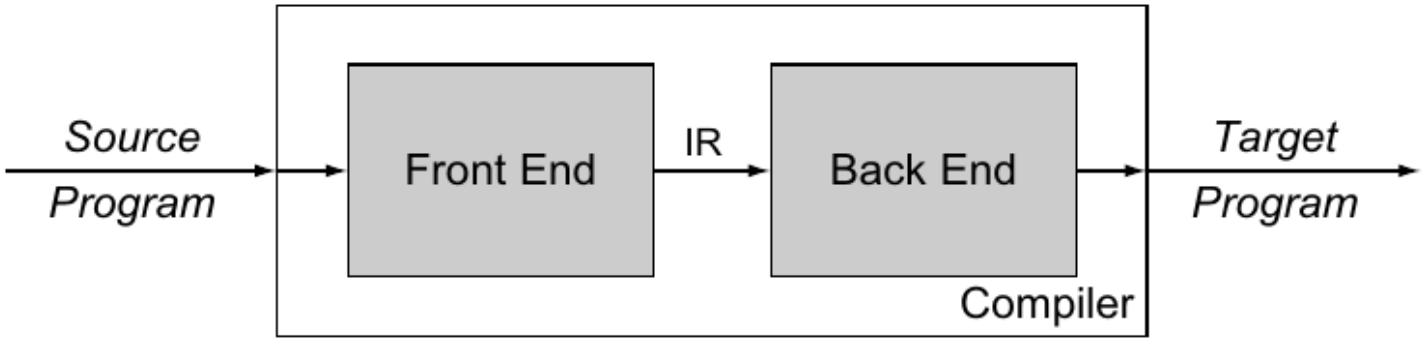
- As operações de análise do compilador inspecionam o programa fonte e determinam as propriedades deste.
- Envolve as fases de análise léxica, sintática e semântica, bem como algumas otimizações de código fonte.
- Podem ser agrupadas em um bloco organizacional chamado *front-end* do compilador.
- Etapas bem definidas e compreendidas: definições formais e algoritmos bem estabelecidos.

Síntese:

- Corresponde à geração de códigos traduzidos.
- Envolve as fases de geração de código e otimizações de código alvo.
- Podem ser agrupadas em um bloco organizacional chamado *back-end* do compilador.
- Soluções são mais especializadas: envolvem heurísticas.

Front-end e *Back-end*

- As partes de análise e síntese podem ser desenvolvidas de forma independente.
- Para tal, é necessário definir uma representação intermediária de código (*intermediate representation – IR*).
- A IR é o meio de comunicação entre *front-end* e *back-end* do compilador.



Separação entre *Front-end* e *Back-end*

- Separação é importante para a portabilidade do compilador e reuso de código.
- Suponha uma suíte de compilação com x linguagens fonte e y arquiteturas alvo.
- Organização monolítica exigiria a implementação de $x \cdot y$ compiladores distintos.
- Por outro lado, se todos os *front-* e *back-ends* entendem a mesma IR ...
- $\Rightarrow$ Organização modular exige somente a implementação de $x + y$ módulos.
- *Exemplo*: No GCC, temos $x > 10$ e $y > 45$.
  $\Rightarrow$ Seria impossível contemplar tantas linguagens e arquiteturas sem uma organização modular.

Passadas

- As repetições realizadas para processar o programa fonte até a geração de código são chamadas de passadas (*passes*).
- Passadas podem ou não corresponder a fases.
- Uma passada pode consistir de várias fases (módulos).
- Maioria dos compiladores usa mais de uma passada. *Ex.:*
  - Uma passada para *scanning* e *parsing*.
  - Uma passada para análise semântica e otimização de código fonte.
  - Uma passada para geração de código e otimização de código alvo.
- Dependendo da linguagem, é possível escrever um compilador *one-pass*.
- Deixa o processo de compilação bastante rápido mas gera código alvo menos otimizado.
- Dada a velocidade dos computadores atuais, compiladores *multi-pass* são a norma.

Bootstrapping e *Porting*

- Compiladores são programas $\Rightarrow$ precisam ser escritos em alguma LP e compilados.
- Dependência circular! Como os compiladores são criados?
- Primeiro montador: escrito em linguagem de máquina.
- Primeiro compilador (CC_1): escrito em linguagem *assembly*.
- Represente CC_1 como $A \leftarrow X \rightarrow B$.
  - A é a linguagem fonte.
  - B é a linguagem alvo.
  - X é a arquitetura aonde o compilador roda.
- Podemos usar CC_1 para construir $CC_2 : C \leftarrow A \rightarrow D$.
- CC_2 é escrito na linguagem A e compilado com CC_1.
- Repetir esse processo *ad nauseam*.

Bootstrapping e *Porting*

Porting:

- Seja $CC_1 : A \leftarrow X \rightarrow B$ um compilador qualquer.
- Se $B \neq X$, CC_1 é chamado de *cross-compiler*.
- Se usarmos CC_1 para compilar $CC_2 : A \leftarrow B \rightarrow B$ dizemos que foi feito um *port* da linguagem A para a arquitetura B.

Bootstrapping:

- É possível que um compilador seja escrito na própria linguagem que ele compila!
- Seja $CC_1 : A \leftarrow X \rightarrow B$ um compilador "*quick-and-dirty*" escrito em linguagem *assembly*.
- Seja $CC_2 : A \leftarrow A \rightarrow B$ um compilador eficiente escrito na sua própria linguagem fonte.
- O processo de usar CC_1 para compilar CC_2 é dito *bootstrapping*.

Aula 00 – Introdução

Prof. Eduardo Zambon

Departamento de Informática (DI)
Centro Tecnológico (CT)
Universidade Federal do Espírito Santo (Ufes)

Compiladores
***Compiler Construction* (CC)**