Um compilador simples de uma passagem

Um tradutor para expressões simples

Prof. Edson Alves

Faculdade UnB Gama

Sumário

1. Um tradutor para expressões simples
2. Análise Léxica
3. Tabela de símbolos
4. Máquinas de Pilha Abstratas
5. Código completo do tradutor

Tradutor dirigido pela sintaxe

- Quando não há associação de atributos aos não-terminais, um tradutor dirigido pela sintaxe pode ser construído a partir da extensão de um analisador gramatical preditivo

Tradutor dirigido pela sintaxe

- Quando não há associação de atributos aos não-terminais, um tradutor dirigido pela sintaxe pode ser construído a partir da extensão de um analisador gramatical preditivo
- Para isso, inicialmente construa o analisador gramatical preditivo

Tradutor dirigido pela sintaxe

- Quando não há associação de atributos aos não-terminais, um tradutor dirigido pela sintaxe pode ser construído a partir da extensão de um analisador gramatical preditivo
- Para isso, inicialmente construa o analisador gramatical preditivo
- Em seguida, copie as ações sintáticas do tradutor nas posições adequadas no analisador gramatical preditivo

Tradutor dirigido pela sintaxe

- Quando não há associação de atributos aos não-terminais, um tradutor dirigido pela sintaxe pode ser construído a partir da extensão de um analisador gramatical preditivo
- Para isso, inicialmente construa o analisador gramatical preditivo
- Em seguida, copie as ações sintáticas do tradutor nas posições adequadas no analisador gramatical preditivo
- Se a gramática tiver uma ou mais produções recursivas à esquerda, é preciso modificar a gramatica para eliminar esta recursão antes de proceder com a construção do analisador gramatical preditivo

Transformação de produções recursivas à esquerda

Transformação de produção recursiva à esquerda

Seja $A \rightarrow A\alpha \mid A\beta \mid \gamma$ uma produção recursiva à esquerda. Esta produção equivale às produções recursivas à direta

$$A \rightarrow \gamma R$$
$$R \rightarrow \alpha R \mid \beta R \mid \epsilon$$

onde α e β é uma cadeia de terminais e não-terminais que não começam com A e nem terminam com R.

Exemplo de transformação de produção recursiva à esquerda

$$expr \rightarrow expr + digito$$
$$expr \rightarrow expr - digito$$
$$expr \rightarrow digito$$

Exemplo de transformação de produção recursiva à esquerda

$$expr \rightarrow expr + digito$$
$$expr \rightarrow expr - digito$$
$$expr \rightarrow digito$$

$$A = expr$$
$$\alpha = + \ digito$$
$$\beta = - \ digito$$
$$\gamma = digito$$
$$R = resto$$

Exemplo de transformação de produção recursiva à esquerda

$$expr \rightarrow expr + digito$$
$$expr \rightarrow expr - digito$$
$$expr \rightarrow digito$$

$$A = expr$$
$$\alpha = + \ digito$$
$$\beta = - \ digito$$
$$\gamma = digito$$
$$R = resto$$

$$expr \rightarrow digito \ resto$$
$$resto \rightarrow + \ digito \ resto$$
$$resto \rightarrow - \ digito \ resto$$
$$resto \rightarrow \epsilon$$

Esquema de tradução da gramática para expressões para a notação posfixa

$$
\begin{aligned}
expr &\rightarrow digito\ resto \\
\\
resto &\rightarrow +\ digito\ \{imprimir(\texttt{'+'})\}\ resto \\
resto &\rightarrow -\ digito\ \{imprimir(\texttt{'-'})\}\ resto \\
resto &\rightarrow \epsilon \\
\\
digito &\rightarrow 0\ \{imprimir(\texttt{'0'})\} \\
digito &\rightarrow 1\ \{imprimir(\texttt{'1'})\} \\
&\ldots \\
digito &\rightarrow 9\ \{imprimir(\texttt{'9'})\}
\end{aligned}
$$

Rotina de extração do próximo token em C++

```
#include <iostream>

using token = char;

token proximo_token()
{
    auto t = std::cin.get();

    return (token) t;
}
```

Rotina de tratamento de erro e declaração de *lookahead* em C++

```
12 token lookahead;
13
14 void erro()
15 {
16     std::cerr << "\nErro de sintaxe! lookahead = " << lookahead << '\n';
17     exit(-1);
18 }
```

Rotina de reconhecimento de tokens em C++

```
20 void reconhecer(token t)
21 {
22     if (lookahead == t)
23         lookahead = proximo_token();
24     else
25         erro();
26 }
```

Rotina associada ao não-terminal *expr* em C++

```
52 void expr()
53 {
54     digito();
55     resto();
56 }
```

Rotina associada ao não-terminal *digito* em C++

```
28 void digito()
29 {
30     if (isdigit(lookahead))
31     {
32         std::cout.put(lookahead);
33         reconhecer(lookahead);
34     } else
35     {
36         erro();
37     }
38 }
```

Rotina associada ao não-terminal *resto* em C++

```
40 void resto()
41 {
42     if (lookahead == '+' or lookahead == '-')
43     {
44         auto c = lookahead;
45         reconhecer(c);
46         digito();
47         std::cout.put(c);
48         resto();
49     }
50 }
```

Rotina principal do tradutor em C++

```
58 int main()
59 {
60     lookahead = proximo_token();
61
62     expr();
63
64     std::cout.put('\n');
65
66     return 0;
67 }
```

Scanners

- Em uma dada gramática, as sentenças de uma linguagem são compostas por cadeias de tokens

Scanners

- Em uma dada gramática, as sentenças de uma linguagem são compostas por cadeias de tokens
- A sequência de caracteres que compõem um único token é denominada lexema

Scanners

- Em uma dada gramática, as sentenças de uma linguagem são compostas por cadeias de tokens
- A sequência de caracteres que compõem um único token é denominada lexema
- Um *scanner* (ou analisador léxico) processa a entrada para produzir uma sequência de tokens

Scanners

- Em uma dada gramática, as sentenças de uma linguagem são compostas por cadeias de tokens
- A sequência de caracteres que compõem um único token é denominada lexema
- Um *scanner* (ou analisador léxico) processa a entrada para produzir uma sequência de tokens
- Dentre as diferentes tarefas que um *scanner* pode realizar estão: remoção de espaços em branco e comentários, identificação de constantes, identificadores e palavras-chave

Remoção de espaços em branco e comentários

- No fluxo de entrada, a presença de outros caracteres que não fazem parte da gramática pode levar a erros no tradutor

Remoção de espaços em branco e comentários

- No fluxo de entrada, a presença de outros caracteres que não fazem parte da gramática pode levar a erros no tradutor
- Várias linguagens permite a presença de "espaços em branco" (espaço em branco, nova linha, tabulação, etc) entre os tokens

Remoção de espaços em branco e comentários

- No fluxo de entrada, a presença de outros caracteres que não fazem parte da gramática pode levar a erros no tradutor
- Várias linguagens permite a presença de "espaços em branco" (espaço em branco, nova linha, tabulação, etc) entre os tokens
- Os espaços em branco podem ser tratados de duas maneiras:

Remoção de espaços em branco e comentários

- No fluxo de entrada, a presença de outros caracteres que não fazem parte da gramática pode levar a erros no tradutor
- Várias linguagens permite a presença de "espaços em branco" (espaço em branco, nova linha, tabulação, etc) entre os tokens
- Os espaços em branco podem ser tratados de duas maneiras:
  1. a gramática deve ser alterada para contemplar os espaços (localização, quantidade, etc), o que traz dificuldades para a especificação da gramática e para a implementação do *scanner*

Remoção de espaços em branco e comentários

- No fluxo de entrada, a presença de outros caracteres que não fazem parte da gramática pode levar a erros no tradutor
- Várias linguagens permite a presença de "espaços em branco" (espaço em branco, nova linha, tabulação, etc) entre os tokens
- Os espaços em branco podem ser tratados de duas maneiras:
  1. a gramática deve ser alterada para contemplar os espaços (localização, quantidade, etc), o que traz dificuldades para a especificação da gramática e para a implementação do *scanner*
  2. o *scanner* simplesmente ignora os espaços em branco (solução mais comum)

Remoção de espaços em branco e comentários

- No fluxo de entrada, a presença de outros caracteres que não fazem parte da gramática pode levar a erros no tradutor
- Várias linguagens permite a presença de "espaços em branco" (espaço em branco, nova linha, tabulação, etc) entre os tokens
- Os espaços em branco podem ser tratados de duas maneiras:
  1. a gramática deve ser alterada para contemplar os espaços (localização, quantidade, etc), o que traz dificuldades para a especificação da gramática e para a implementação do *scanner*
  2. o *scanner* simplesmente ignora os espaços em branco (solução mais comum)
- O *scanner* também pode ignorar o comentários, de modo que estes possam ser tratados como espaços em branco

Identificação de constantes inteiras

- Constantes inteiras são sequências de dígitos

Identificação de constantes inteiras

- Constantes inteiras são sequências de dígitos
- As constantes podem ser inseridas na gramática da linguagem por meio de produções, ou sua identificação pode ser delegada para o analisador léxico, que irá criar tokens para estas constantes

Identificação de constantes inteiras

- Constantes inteiras são sequências de dígitos
- As constantes podem ser inseridas na gramática da linguagem por meio de produções, ou sua identificação pode ser delegada para o analisador léxico, que irá criar tokens para estas constantes
- A segunda alternativa permite tratar constantes inteiras como unidades autônomas durante a tradução

Identificação de constantes inteiras

- Constantes inteiras são sequências de dígitos
- As constantes podem ser inseridas na gramática da linguagem por meio de produções, ou sua identificação pode ser delegada para o analisador léxico, que irá criar tokens para estas constantes
- A segunda alternativa permite tratar constantes inteiras como unidades autônomas durante a tradução
- Para cada constante inteira, o *scanner* gerará um token e um atributo, sendo o token um identificador de constantes inteiras (por exemplo, `num`) e o atributo o valor inteiro da constante

Identificação de constantes inteiras

- Constantes inteiras são sequências de dígitos
- As constantes podem ser inseridas na gramática da linguagem por meio de produções, ou sua identificação pode ser delegada para o analisador léxico, que irá criar tokens para estas constantes
- A segunda alternativa permite tratar constantes inteiras como unidades autônomas durante a tradução
- Para cada constante inteira, o *scanner* gerará um token e um atributo, sendo o token um identificador de constantes inteiras (por exemplo, `num`) e o atributo o valor inteiro da constante
- Por exemplo, a entrada `3 + 14 + 15` seria transformada na sequência de tokens

  `<num, 3>` `<+,>` `<num, 14>` `<+,>` `<num, 15>`

  onde o par `<x, y>` indica que o token `x` tem atributo `y`

Reconhecimento de identificadores e palavras-chave

- As linguagens de programação utilizam identificadores para nomear variáveis, vetores, funções e outros elementos

Reconhecimento de identificadores e palavras-chave

- As linguagens de programação utilizam identificadores para nomear variáveis, vetores, funções e outros elementos
- As gramáticas das linguagens, em geral, tratam os identificadores como tokens

Reconhecimento de identificadores e palavras-chave

- As linguagens de programação utilizam identificadores para nomear variáveis, vetores, funções e outros elementos
- As gramáticas das linguagens, em geral, tratam os identificadores como tokens
- Os analisadores gramaticais (*parsers*) destas gramáticas esperam um mesmo token (por exemplo, **id**) sempre que um identificador aparece na entrada

Reconhecimento de identificadores e palavras-chave

- As linguagens de programação utilizam identificadores para nomear variáveis, vetores, funções e outros elementos
- As gramáticas das linguagens, em geral, tratam os identificadores como tokens
- Os analisadores gramaticais (*parsers*) destas gramáticas esperam um mesmo token (por exemplo, **id**) sempre que um identificador aparece na entrada
- Por exemplo, a expressão `x = x + y;` deve ser convertida pelo *scanner* para

id = id + id ;

Reconhecimento de identificadores e palavras-chave

- As linguagens de programação utilizam identificadores para nomear variáveis, vetores, funções e outros elementos
- As gramáticas das linguagens, em geral, tratam os identificadores como tokens
- Os analisadores gramaticais (*parsers*) destas gramáticas esperam um mesmo token (por exemplo, **id**) sempre que um identificador aparece na entrada
- Por exemplo, a expressão `x = x + y;` deve ser convertida pelo *scanner* para

  id = **id** + **id**;

- Na análise sintática, é útil saber que as duas primeiras ocorrências de **id** se referem ao lexema `x`, enquanto que a última se refere ao lexema `y`

Reconhecimento de identificadores e palavras-chave

- Uma tabela de símbolos pode ser usada para determinar se um dado lexema já foi encontrado ou não

Reconhecimento de identificadores e palavras-chave

- Uma tabela de símbolos pode ser usada para determinar se um dado lexema já foi encontrado ou não
- Na primeira ocorrência o lexema é armazenado na tabela de símbolos e também nela, e em todas as demais ocorrências, o lexema se torna o atributo do token **id**

Reconhecimento de identificadores e palavras-chave

- Uma tabela de símbolos pode ser usada para determinar se um dado lexema já foi encontrado ou não
- Na primeira ocorrência o lexema é armazenado na tabela de símbolos e também nela, e em todas as demais ocorrências, o lexema se torna o atributo do token **id**
- As palavras-chave da linguagem são cadeias fixas de caracteres usadas como pontuação ou para identificar determinadas construções

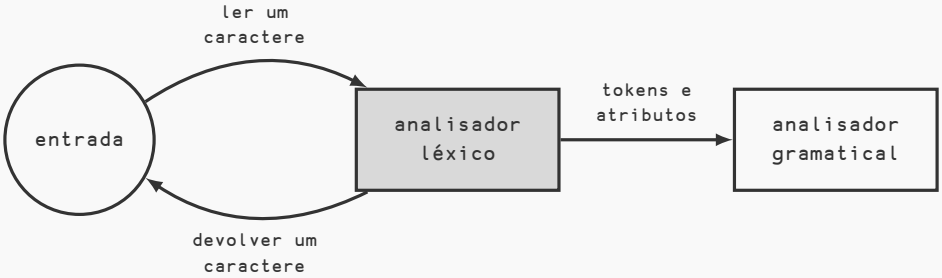
Reconhecimento de identificadores e palavras-chave

- Uma tabela de símbolos pode ser usada para determinar se um dado lexema já foi encontrado ou não
- Na primeira ocorrência o lexema é armazenado na tabela de símbolos e também nela, e em todas as demais ocorrências, o lexema se torna o atributo do token **id**
- As palavras-chave da linguagem são cadeias fixas de caracteres usadas como pontuação ou para identificar determinadas construções
- Em geral, as palavras-chave seguem a mesma regra de formação dos identificadores

Reconhecimento de identificadores e palavras-chave

- Uma tabela de símbolos pode ser usada para determinar se um dado lexema já foi encontrado ou não
- Na primeira ocorrência o lexema é armazenado na tabela de símbolos e também nela, e em todas as demais ocorrências, o lexema se torna o atributo do token **id**
- As palavras-chave da linguagem são cadeias fixas de caracteres usadas como pontuação ou para identificar determinadas construções
- Em geral, as palavras-chave seguem a mesma regra de formação dos identificadores
- Se as palavras-chave forem reservadas, isto é, não puderem ser usadas como identificadores, a situação fica facilitada: um lexema só será um identificador caso não seja uma palavra-chave

Interface para um analisador léxico



Produtor e consumidor

- O analisador léxico e o analisador gramatical formam um par produtor-consumidor

Produtor e consumidor

- O analisador léxico e o analisador gramatical formam um par produtor-consumidor
- O analisador léxico produz tokens; o analisador gramatical os consome

Produtor e consumidor

- O analisador léxico e o analisador gramatical formam um par produtor-consumidor
- O analisador léxico produz tokens; o analisador gramatical os consome
- A interação entre ambos depende do *buffer* que armazena os tokens produzidos: o *scanner* não pode gerar novos tokens se o *buffer* está cheio, o *parser* não pode prosseguir se o *buffer* estiver vazio

Produtor e consumidor

- O analisador léxico e o analisador gramatical formam um par produtor-consumidor
- O analisador léxico produz tokens; o analisador gramatical os consome
- A interação entre ambos depende do *buffer* que armazena os tokens produzidos: o *scanner* não pode gerar novos tokens se o *buffer* está cheio, o *parser* não pode prosseguir se o *buffer* estiver vazio
- Em geral, o *buffer* armazena um único token

Produtor e consumidor

- O analisador léxico e o analisador gramatical formam um par produtor-consumidor
- O analisador léxico produz tokens; o analisador gramatical os consome
- A interação entre ambos depende do *buffer* que armazena os tokens produzidos: o *scanner* não pode gerar novos tokens se o *buffer* está cheio, o *parser* não pode prosseguir se o *buffer* estiver vazio
- Em geral, o *buffer* armazena um único token
- Neste caso, o *parser* pode requisitar ao *scanner*, por demanda, a produção de novos tokens

Implementação da identificação de constantes inteiras

- Para que as constantes inteiras possam ser devidamente identificadas no código do *scanner*, é preciso que elas façam parte da gramática

Implementação da identificação de constantes inteiras

- Para que as constantes inteiras possam ser devidamente identificadas no código do *scanner*, é preciso que elas façam parte da gramática
- Por exemplo, a produção do não terminal $fator$

$$fator \rightarrow \textbf{digito} \mid (expr)$$

pode ser modificada para

$$fator \rightarrow (expr) \mid \textbf{num} \quad \{imprimir(\textbf{num}.valor)\}$$

Implementação da identificação de constantes inteiras

- Para que as constantes inteiras possam ser devidamente identificadas no código do *scanner*, é preciso que elas façam parte da gramática
- Por exemplo, a produção do não terminal $fator$

$$fator \rightarrow \textbf{digito} \mid (expr)$$

pode ser modificada para

$$fator \rightarrow (expr) \mid \textbf{num} \quad \{imprimir(\textbf{num}.valor)\}$$

- Em relação à implementação, um token deve ser identificado por um par contendo o identificador do token e o seu atributo

Exemplo de implementação do terminal *fator* em C++

```
using token_t = std::pair<int, int>;

// NUM deve ter um valor diferente de qualquer caractere da tabela ASCII
const int NUM { 256 };

void fator()
{
    auto [token, valor] = lookahead;

    if (token == '(') {
        reconhecer('(');
        expr();
        reconhecer(')');
    } else if (token == NUM) {
        reconhecer(NUM);
        std::cout << valor;
    } else
        erro();
}
```

Exemplo de implementação de um scanner de constantes inteiras em C++

```
1 #include <bits/stdc++.h>
2
3 using token_t = std::pair<int, int>;
4 const int NUM = 256, NONE = -1;
5
6 token_t scanner()
7 {
8     while (not std::cin.eof())
9     {
10        auto c = std::cin.get();
11
12        if (isspace(c))
13            continue;
```

Exemplo de implementação de um scanner de constantes inteiras em C++

```
15          if (isdigit(c))
16          {
17              int valor = c - '0';
18
19              while (not std::cin.eof() and (c = std::cin.get(), isdigit(c)))
20                  valor = 10*valor + (c - '0');
21
22              std::cin.unget();
23
24              return { NUM, valor };
25          } else
26              return { c, NONE };
27      }
28
29      return { EOF, NONE };
30  }
```

Exemplo de implementação de um scanner de constantes inteiras em C++

```
32 int main()
33 {
34     while (true)
35     {
36         auto [lookahead, valor] = scanner();
37
38         if (lookahead == EOF)
39             break;
40         else if (lookahead == NUM)
41             std::cout << "Número lido: " << valor << '\n';
42         else
43             std::cout << "Token lido: " << (char) lookahead << '\n';
44     }
45
46     return 0;
47 }
```

Tabela de símbolos

- Uma tabela de símbolos é uma estrutura de dados que armazena informações nas diferentes fases da compilação

Tabela de símbolos

- Uma tabela de símbolos é uma estrutura de dados que armazena informações nas diferentes fases da compilação
- As fases de análise coletam informações que são usadas nas fases de síntese para a geração do programa alvo

Tabela de símbolos

- Uma tabela de símbolos é uma estrutura de dados que armazena informações nas diferentes fases da compilação
- As fases de análise coletam informações que são usadas nas fases de síntese para a geração do programa alvo
- Por exemplo, na análise léxica os lexemas são adicionados à tabela de símbolos

Tabela de símbolos

- Uma tabela de símbolos é uma estrutura de dados que armazena informações nas diferentes fases da compilação
- As fases de análise coletam informações que são usadas nas fases de síntese para a geração do programa alvo
- Por exemplo, na análise léxica os lexemas são adicionados à tabela de símbolos
- As fases posteriores podem adicionar informações a este lexema, como tipo, uso (procedimento, variável, etc) e posição no armazenamento

A análise léxica e a tabela de símbolos

- As principais interações que ocorrem com a tabela de símbolos na análise léxica tratam do armazenamento e recuperação de lexemas

A análise léxica e a tabela de símbolos

- As principais interações que ocorrem com a tabela de símbolos na análise léxica tratam do armazenamento e recuperação de lexemas
- A rotina $\text{INSERIR}(s, t)$ insere o lexema s na tabela, sendo ele associado ao token t

A análise léxica e a tabela de símbolos

- As principais interações que ocorrem com a tabela de símbolos na análise léxica tratam do armazenamento e recuperação de lexemas
- A rotina $\text{INSERIR}(s, t)$ insere o lexema s na tabela, sendo ele associado ao token t
- A rotina $\text{BUSCAR}(s)$ regasta o lexema s, permitindo a consulta (e, se necessário, atualização) de seu token

A análise léxica e a tabela de símbolos

- As principais interações que ocorrem com a tabela de símbolos na análise léxica tratam do armazenamento e recuperação de lexemas
- A rotina $\text{INSERIR}(s, t)$ insere o lexema s na tabela, sendo ele associado ao token t
- A rotina $\text{BUSCAR}(s)$ regasta o lexema s, permitindo a consulta (e, se necessário, atualização) de seu token
- Em geral a operação de busca é usada para saber se um determinado lexema já está na tabela ou não

A análise léxica e a tabela de símbolos

- As principais interações que ocorrem com a tabela de símbolos na análise léxica tratam do armazenamento e recuperação de lexemas
- A rotina $\text{INSERIR}(s, t)$ insere o lexema s na tabela, sendo ele associado ao token t
- A rotina $\text{BUSCAR}(s)$ regasta o lexema s, permitindo a consulta (e, se necessário, atualização) de seu token
- Em geral a operação de busca é usada para saber se um determinado lexema já está na tabela ou não
- Em linguagem que possuem dicionários em sua biblioteca padrão, a tabela pode ser implementada por meio de um dicionário

Tabela de símbolos e palavras reservadas

- As rotinas descritas para a tabela de símbolos permitem um tratamento direto de quaisquer palavras reservadas da linguagem

Tabela de símbolos e palavras reservadas

- As rotinas descritas para a tabela de símbolos permitem um tratamento direto de quaisquer palavras reservadas da linguagem
- Por exemplo, considere que **div** e **mod** são palavras reservadas

Tabela de símbolos e palavras reservadas

- As rotinas descritas para a tabela de símbolos permitem um tratamento direto de quaisquer palavras reservadas da linguagem
- Por exemplo, considere que **div** e **mod** são palavras reservadas
- A rotina de inserção permite associar os lexemas `"div"` e `"mod"` aos tokens **div** e **mod**:

$$\text{INSERIR}(\texttt{"div"}, \textbf{div})$$
$$\text{INSERIR}(\texttt{"mod"}, \textbf{mod})$$

Tabela de símbolos e palavras reservadas

- As rotinas descritas para a tabela de símbolos permitem um tratamento direto de quaisquer palavras reservadas da linguagem
- Por exemplo, considere que **div** e **mod** são palavras reservadas
- A rotina de inserção permite associar os lexemas `"div"` e `"mod"` aos tokens **div** e **mod**:

$$\text{INSERIR}(\texttt{"div"}, \textbf{div})$$
$$\text{INSERIR}(\texttt{"mod"}, \textbf{mod})$$

- Qualquer chamada posterior da rotina de busca retornará os tokens, de modo que os lexemas já não poderão ser mais usados como identificadores

Pseudocódigo de um analisador léxico que manipula identificadores

```
1:  function SCANNER( )
2:      loop
3:          c ← PROXIMOCARACTERE( )
4:          if c é espaço then
5:              reinicie o laço
6:          else if c é um dígito then
7:              v ← LERCONSTANTEINTEIRA( )
8:              return { NUM, v }
9:          else if c é uma letra then
10:             lexema ← LERPALAVRA( )
11:             if lexema não está na tabela de símbolos then
12:                 INSERIR(lexema, ID)
13:             return { OBTERTOKEN(lexema), lexema }
14:         else
15:             return { c, NONE }
```

Máquinas de Pilha Abstratas

- A interface de vanguarda do compilador produz uma representação intermediária do programa fonte, que será usada pela interface de retaguarda para produzir o programa alvo

Máquinas de Pilha Abstratas

- A interface de vanguarda do compilador produz uma representação intermediária do programa fonte, que será usada pela interface de retaguarda para produzir o programa alvo
- Uma possível forma para a representação intermediária é a máquina de pilha abstrata

Máquinas de Pilha Abstratas

- A interface de vanguarda do compilador produz uma representação intermediária do programa fonte, que será usada pela interface de retaguarda para produzir o programa alvo
- Uma possível forma para a representação intermediária é a máquina de pilha abstrata
- Uma máquina de pilha abstrata possui memórias separadas para dados e instruções, e todas as operações aritméticas são realizadas sobre os valores em uma pilha

Máquinas de Pilha Abstratas

- A interface de vanguarda do compilador produz uma representação intermediária do programa fonte, que será usada pela interface de retaguarda para produzir o programa alvo
- Uma possível forma para a representação intermediária é a máquina de pilha abstrata
- Uma máquina de pilha abstrata possui memórias separadas para dados e instruções, e todas as operações aritméticas são realizadas sobre os valores em uma pilha
- As instruções são divididas em três classes: aritmética inteira, manipulação de pilha e fluxo de controle

Máquinas de Pilha Abstratas

- A interface de vanguarda do compilador produz uma representação intermediária do programa fonte, que será usada pela interface de retaguarda para produzir o programa alvo
- Uma possível forma para a representação intermediária é a máquina de pilha abstrata
- Uma máquina de pilha abstrata possui memórias separadas para dados e instruções, e todas as operações aritméticas são realizadas sobre os valores em uma pilha
- As instruções são divididas em três classes: aritmética inteira, manipulação de pilha e fluxo de controle
- O ponteiro pc indica qual é a próxima instrução a ser executada

Instruções aritméticas

- A máquina de pilha abstrata precisa implementar cada operador da linguagem intermediária

Instruções aritméticas

- A máquina de pilha abstrata precisa implementar cada operador da linguagem intermediária
- Operações elementares, como adição e subtração, são suportadas diretamente

Instruções aritméticas

- A máquina de pilha abstrata precisa implementar cada operador da linguagem intermediária
- Operações elementares, como adição e subtração, são suportadas diretamente
- Operações mais sofisticadas devem ser implementadas como uma sequência de instruções da máquina

Instruções aritméticas

- A máquina de pilha abstrata precisa implementar cada operador da linguagem intermediária
- Operações elementares, como adição e subtração, são suportadas diretamente
- Operações mais sofisticadas devem ser implementadas como uma sequência de instruções da máquina
- A título de simplificação, assuma que existe uma instrução para cada operação aritmética

Instruções aritméticas

- A máquina de pilha abstrata precisa implementar cada operador da linguagem intermediária
- Operações elementares, como adição e subtração, são suportadas diretamente
- Operações mais sofisticadas devem ser implementadas como uma sequência de instruções da máquina
- A título de simplificação, assuma que existe uma instrução para cada operação aritmética
- O código de uma máquina de pilha abstrata para uma expressão simula a avaliação de um representação posfixa, usando uma pilha

Instruções aritméticas

- A máquina de pilha abstrata precisa implementar cada operador da linguagem intermediária
- Operações elementares, como adição e subtração, são suportadas diretamente
- Operações mais sofisticadas devem ser implementadas como uma sequência de instruções da máquina
- A título de simplificação, assuma que existe uma instrução para cada operação aritmética
- O código de uma máquina de pilha abstrata para uma expressão simula a avaliação de um representação posfixa, usando uma pilha
- A avaliação segue da esquerda para a direita, empilhando os operandos

Instruções aritméticas

- A máquina de pilha abstrata precisa implementar cada operador da linguagem intermediária
- Operações elementares, como adição e subtração, são suportadas diretamente
- Operações mais sofisticadas devem ser implementadas como uma sequência de instruções da máquina
- A título de simplificação, assuma que existe uma instrução para cada operação aritmética
- O código de uma máquina de pilha abstrata para uma expressão simula a avaliação de um representação posfixa, usando uma pilha
- A avaliação segue da esquerda para a direita, empilhando os operandos
- Quando um operador é encontrado, seus operandos são extraídos da pilha (do último para o primeiro), a operação é realizada e o resultado é inserido no topo da pilha

Exemplo de avaliação da expressão `12+3*` por máquina abstrata de pilha

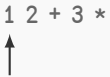
Instrução **Ações**

Pilha

Exemplo de avaliação da expressão `12+3*` por máquina abstrata de pilha

Instrução

Ações



Pilha

Exemplo de avaliação da expressão `12+3*` por máquina abstrata de pilha

Instrução

```
1 2 + 3 *
↑
```

Ações

```
Empilhar o valor 1
```

Pilha

Exemplo de avaliação da expressão `12+3*` por máquina abstrata de pilha

Instrução

```
1 2 + 3 *
↑
```

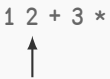
Ações

```
Empilhar o valor 1
```

Pilha

Exemplo de avaliação da expressão `12+3*` por máquina abstrata de pilha

Instrução **Ações**



Pilha

Exemplo de avaliação da expressão `12+3*` por máquina abstrata de pilha

Instrução

```
1 2 + 3 *
  ↑
```

Ações

```
Empilhar o valor 2
```

Pilha

Exemplo de avaliação da expressão `12+3*` por máquina abstrata de pilha

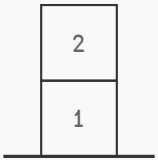
Instrução

```
1 2 + 3 *
  ↑
```

Ações

```
Empilhar o valor 2
```

Pilha

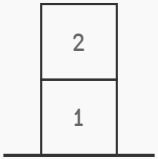


Exemplo de avaliação da expressão `12+3*` por máquina abstrata de pilha

Instrução **Ações**

`1 2 + 3 *`

Pilha



Exemplo de avaliação da expressão `12+3*` por máquina abstrata de pilha

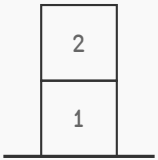
Instrução

```
1 2 + 3 *
  ↑
```

Ações

`Adicionar 1+2`

Pilha



Exemplo de avaliação da expressão 12+3* por máquina abstrata de pilha

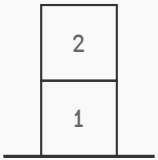
Instrução

```
1 2 + 3 *
  ↑
```

Ações

```
Adicionar 1+2
Empilhar a soma
```

Pilha



Exemplo de avaliação da expressão `12+3*` por máquina abstrata de pilha

Instrução

```
1 2 + 3 *
  ↑
```

Ações

```
Adicionar 1+2
Empilhar a soma
```

Pilha

Exemplo de avaliação da expressão `12+3*` por máquina abstrata de pilha

Instrução

```
1 2 + 3 *
    ↑
```

Ações

```
Empilhar o valor 3
```

Pilha

Exemplo de avaliação da expressão `12+3*` por máquina abstrata de pilha

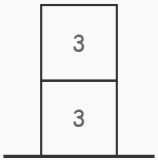
Instrução

```
1 2 + 3 *
    ↑
```

Ações

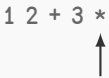
```
Empilhar o valor 3
```

Pilha

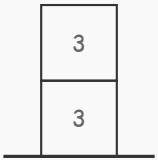


Exemplo de avaliação da expressão `12+3*` por máquina abstrata de pilha

Instrução **Ações**



Pilha



Exemplo de avaliação da expressão 12+3* por máquina abstrata de pilha

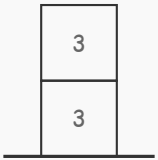
Instrução

```
1 2 + 3 *
      ↑
```

Ações

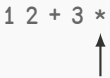
```
Multiplicar 3*3
```

Pilha



Exemplo de avaliação da expressão `12+3*` por máquina abstrata de pilha

Instrução

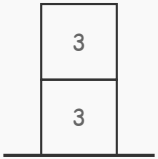


Ações

`Multiplicar 3*3`

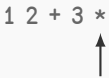
`Empilhar o produto`

Pilha



Exemplo de avaliação da expressão `12+3*` por máquina abstrata de pilha

Instrução



Ações

`Multiplicar 3*3`

`Empilhar o produto`

Pilha

Valores-L e valores-R

- O significado de um identificador depende da posição onde ele ocorre em uma atribuição

Valores-L e valores-R

- O significado de um identificador depende da posição onde ele ocorre em uma atribuição
- No lado esquerdo, o identificador se refere à localização de memória onde o valor deve ser armazenado

Valores-L e valores-R

- O significado de um identificador depende da posição onde ele ocorre em uma atribuição
- No lado esquerdo, o identificador se refere à localização de memória onde o valor deve ser armazenado
- No lado direito, o identificador se refere ao valor armazenado na localização de memória associada ao identificador

Valores-L e valores-R

- O significado de um identificador depende da posição onde ele ocorre em uma atribuição
- No lado esquerdo, o identificador se refere à localização de memória onde o valor deve ser armazenado
- No lado direito, o identificador se refere ao valor armazenado na localização de memória associada ao identificador
- Valor-L e valor-R se referem aos valores apropriados para os lados esquerdo e direito de uma atribuição, respectivamente

Valores-L e valores-R

- O significado de um identificador depende da posição onde ele ocorre em uma atribuição
- No lado esquerdo, o identificador se refere à localização de memória onde o valor deve ser armazenado
- No lado direito, o identificador se refere ao valor armazenado na localização de memória associada ao identificador
- Valor-L e valor-R se referem aos valores apropriados para os lados esquerdo e direito de uma atribuição, respectivamente
- Um mesmo identificador pode ser um valor-L e um valor-R na mesma atribuição (por exemplo, o identificador `x` em `x = x + 1`)

Manipulação da pilha

Uma máquina de pilha abstrata suporta as seguintes instruções para a manipulação da pilha:

Instrução	Significado
`push` v	empilha v
`pop`	desempilha o valor do topo da pilha
`valor-r` p	empilha o valor armazenado no endereço de memória p
`valor-l` p	empilha o endereço de memória p
`:=`	o valor-R do topo da pilha é armazenado no valor-L do subtopo (elemento que está abaixo do topo) da pilha
`copiar`	empilha o valor do topo da pilha

Tradução de expressões

- O código para avaliar uma expressão na máquina de pilha abstrata tem uma relação direta com a notação posfixa

Tradução de expressões

- O código para avaliar uma expressão na máquina de pilha abstrata tem uma relação direta com a notação posfixa
- Por exemplo, a expressão `a + b` é traduzida para o código intermediário

```
valor-r   a
valor-r   b
+
```

Tradução de expressões

- O código para avaliar uma expressão na máquina de pilha abstrata tem uma relação direta com a notação posfixa
- Por exemplo, a expressão `a + b` é traduzida para o código intermediário

```
valor-r   a
valor-r   b
+
```

- As atribuições são traduzidas da seguinte maneira: o valor-L do identificador é empilhado, a expressão à direita é avaliada e o seu valor r é atribuído ao identificador

Tradução de expressões

- O código para avaliar uma expressão na máquina de pilha abstrata tem uma relação direta com a notação posfixa
- Por exemplo, a expressão `a + b` é traduzida para o código intermediário

```
valor-r   a
valor-r   b
+
```

- As atribuições são traduzidas da seguinte maneira: o valor-L do identificador é empilhado, a expressão à direita é avaliada e o seu valor r é atribuído ao identificador
- Formalmente,

$$cmd \rightarrow \mathbf{id} := expr \ \{ \ cmd.t := \texttt{"valor-l"} \ || \ \mathbf{id}.lexema \ || \ expr \ || \ \texttt{":="} \ \}$$

Tradução da expressão `F = 9*C/5 + 32` para máquina de pilha abstrata

```
1    valor-l     F
2    push        9
3    valor-r     C
4    *
5    push        5
6    /
7    push        32
8    +
9    :=
```

Fluxo de controle

- A máquina de pilha abstrata executa as instruções sequencialmente, na ordem em que foram dadas

Fluxo de controle

- A máquina de pilha abstrata executa as instruções sequencialmente, na ordem em que foram dadas
- Há três formas de se especificar um desvio (salto) no fluxo de execução

Fluxo de controle

- A máquina de pilha abstrata executa as instruções sequencialmente, na ordem em que foram dadas
- Há três formas de se especificar um desvio (salto) no fluxo de execução
  1. o operando da instrução fornece o endereço da localização alvo;

Fluxo de controle

- A máquina de pilha abstrata executa as instruções sequencialmente, na ordem em que foram dadas
- Há três formas de se especificar um desvio (salto) no fluxo de execução
  1. o operando da instrução fornece o endereço da localização alvo;
  2. o operando da instrução fornece a distância relativa (positiva ou negativa) a ser saltada

Fluxo de controle

- A máquina de pilha abstrata executa as instruções sequencialmente, na ordem em que foram dadas
- Há três formas de se especificar um desvio (salto) no fluxo de execução
  1. o operando da instrução fornece o endereço da localização alvo;
  2. o operando da instrução fornece a distância relativa (positiva ou negativa) a ser saltada
  3. o alvo é especificado simbolicamente (a máquina deve suportar rótulos)

Fluxo de controle

- A máquina de pilha abstrata executa as instruções sequencialmente, na ordem em que foram dadas
- Há três formas de se especificar um desvio (salto) no fluxo de execução
  1. o operando da instrução fornece o endereço da localização alvo;
  2. o operando da instrução fornece a distância relativa (positiva ou negativa) a ser saltada
  3. o alvo é especificado simbolicamente (a máquina deve suportar rótulos)
- Nas duas primeiras formas, uma alternativa é especificar que o salto está no topo da pilha

Fluxo de controle

- A máquina de pilha abstrata executa as instruções sequencialmente, na ordem em que foram dadas
- Há três formas de se especificar um desvio (salto) no fluxo de execução
  1. o operando da instrução fornece o endereço da localização alvo;
  2. o operando da instrução fornece a distância relativa (positiva ou negativa) a ser saltada
  3. o alvo é especificado simbolicamente (a máquina deve suportar rótulos)
- Nas duas primeiras formas, uma alternativa é especificar que o salto está no topo da pilha
- A terceira forma é a mais simples de se implementar, pois não há necessidade de se recalcular os endereços caso o número de instruções seja modificado durante a otimização

Instruções de fluxo de controle

Uma máquina de pilha abstrata suporta as seguintes instruções para o fluxo de controle:

Instrução	Significado
`rótulo` r	especifica o rótulo r como possível alvo de desvios; não há outros efeitos
`goto` r	a próxima instrução é tomada a partir do rótulo r
`gofalse` r	desempilha o topo da pilha: salta para r se o valor for igual a zero
`gotrue` r	desempilha o topo da pilha: salta para r se o valor for diferente de zero
`parar`	encerra a execução

Gabaritos para tradução de condicionais e laços

if

código para *expr*
`gofalse` *saida*
código para *cmd*
`rotulo` *saida*

while

`rotulo` *teste*
código para *expr*
`gofalse` *saida*
código para *cmd*
`goto` *teste*
`rotulo` *saida*

Unicidade dos rótulos

- Os rótulos $saida$ e $teste$ que ilustram os gabaritos das condicionais e dos laços devem ser únicos, para evitar possíveis ambiguidades

Unicidade dos rótulos

- Os rótulos $saida$ e $teste$ que ilustram os gabaritos das condicionais e dos laços devem ser únicos, para evitar possíveis ambiguidades
- É preciso, portanto, de uma estratégia que torne tais rótulos únicos durante a tradução

Unicidade dos rótulos

- Os rótulos $saida$ e $teste$ que ilustram os gabaritos das condicionais e dos laços devem ser únicos, para evitar possíveis ambiguidades
- É preciso, portanto, de uma estratégia que torne tais rótulos únicos durante a tradução
- Seja $novoRotulo$ um procedimento que gera, a cada chamada, um novo rótulo único

Unicidade dos rótulos

- Os rótulos $saida$ e $teste$ que ilustram os gabaritos das condicionais e dos laços devem ser únicos, para evitar possíveis ambiguidades
- É preciso, portanto, de uma estratégia que torne tais rótulos únicos durante a tradução
- Seja $novoRotulo$ um procedimento que gera, a cada chamada, um novo rótulo único
- A ação semântica associada ao comando `if` assumiria a seguinte forma:

$$
\begin{array}{lll}
cmd \rightarrow \texttt{if}\ expr\ \texttt{then}\ cmd_1\ \{ & saida & := novoRotulo; \\
& cmd.t & := expr.t \\
& & \ ||\ \texttt{"gofalse"}\ saida \\
& & \ ||\ cmd_1.t \\
& & \ ||\ \texttt{"rotulo"}\ saida\ \}
\end{array}
$$

Código completo do tradutor

- O código a seguir implementa um tradutor de expressões em forma infixa, terminadas por ponto-e-vírgula, para forma posfixa

Código completo do tradutor

- O código a seguir implementa um tradutor de expressões em forma infixa, terminadas por ponto-e-vírgula, para forma posfixa
- Ele será implementado a partir da tradução dirigida a sintaxe

Código completo do tradutor

- O código a seguir implementa um tradutor de expressões em forma infixa, terminadas por ponto-e-vírgula, para forma posfixa
- Ele será implementado a partir da tradução dirigida a sintaxe
- As expressões conterão números, identificadores e os operadores `+`, `-`, `*`, `/`, `div` e `mod`

Código completo do tradutor

- O código a seguir implementa um tradutor de expressões em forma infixa, terminadas por ponto-e-vírgula, para forma posfixa
- Ele será implementado a partir da tradução dirigida a sintaxe
- As expressões conterão números, identificadores e os operadores `+`, `-`, `*`, `/`, `div` e `mod`
- Os tokens são: **id**, para identificadores, **num**, para constantes inteiras e **eof** para fim de arquivo

Código completo do tradutor

- O código a seguir implementa um tradutor de expressões em forma infixa, terminadas por ponto-e-vírgula, para forma posfixa
- Ele será implementado a partir da tradução dirigida a sintaxe
- As expressões conterão números, identificadores e os operadores `+`, `-`, `*`, `/`, `div` e `mod`
- Os tokens são: **id**, para identificadores, **num**, para constantes inteiras e **eof** para fim de arquivo
- Cada módulo será implementado em um par de arquivos `.cpp` e `.h`, exceto pelo módulo principal (`main.cpp`)

Especificação para um tradutor infixa-posfixa

$$
\begin{array}{lcll}
\mathit{inicio} & \rightarrow & \mathit{lista}\ \textbf{eof} & \\
\mathit{lista} & \rightarrow & \mathit{expr}\ \texttt{;}\ \{\ \mathit{imprimir}(\texttt{'\textbackslash n'})\ \}\ \mathit{lista} & \\
 & | & \epsilon & \\
\mathit{expr} & \rightarrow & \mathit{expr}\ \texttt{+}\ \mathit{termo} & \{\ \mathit{imprimir}(\texttt{'+'})\ \} \\
 & | & \mathit{expr}\ \texttt{-}\ \mathit{termo} & \{\ \mathit{imprimir}(\texttt{'-'})\ \} \\
 & | & \mathit{termo} & \\
\mathit{termo} & \rightarrow & \mathit{termo}\ \texttt{*}\ \mathit{fator} & \{\ \mathit{imprimir}(\texttt{'*'})\ \} \\
 & | & \mathit{termo}\ \texttt{/}\ \mathit{fator} & \{\ \mathit{imprimir}(\texttt{'/'})\ \} \\
 & | & \mathit{termo}\ \textbf{div}\ \mathit{fator} & \{\ \mathit{imprimir}(\texttt{"DIV"})\ \} \\
 & | & \mathit{termo}\ \textbf{mod}\ \mathit{fator} & \{\ \mathit{imprimir}(\texttt{"MOD"})\ \} \\
 & | & \mathit{fator} & \\
\mathit{fator} & \rightarrow & (\mathit{expr}) & \\
 & | & \textbf{id} & \{\ \mathit{imprimir}(\textbf{id}.\mathit{lexema})\ \} \\
 & | & \textbf{num} & \{\ \mathit{imprimir}(\textbf{id}.\mathit{valor})\ \}
\end{array}
$$

Descrição dos tokens

Lexema	Token	Atributo
espaço em branco		
sequência de dígitos	`NUM`	valor númerico da sequência
div	`DIV`	
mod	`MOD`	
sequências iniciada em letra e seguida de letras e dígitos	`ID`	lexema
caractere de fim de arquivo	`DONE`	
qualquer outro caractere	o próprio caractere	

Módulo `main.cpp`

- Este módulo é responsável pela início da execução do programa

Módulo `main.cpp`

- Este módulo é responsável pela início da execução do programa
- Ele simplesmente invoca o tradutor

Módulo `main.cpp`

- Este módulo é responsável pela início da execução do programa
- Ele simplesmente invoca o tradutor

```
1 #include "parser.h"
2
3 int main()
4 {
5     parser::parse();
6
7     return 0;
8 }
```

Módulo token

- Este módulo define uma estrutura para a representação de um token

Módulo token

- Este módulo define uma estrutura para a representação de um token
- Cada token tem um tipo e um atributo associado

Módulo `token`

- Este módulo define uma estrutura para a representação de um token
- Cada token tem um tipo e um atributo associado
- Como o atributo de `NUM` é inteiro e de `ID` é string, foi utilizado o tipo `variant<int, string>` de C++17

Módulo `token`

- Este módulo define uma estrutura para a representação de um token
- Cada token tem um tipo e um atributo associado
- Como o atributo de `NUM` é inteiro e de `ID` é string, foi utilizado o tipo `variant<int, string>` de C++17
- Os tipos dos tokens foram codificados como inteiros, com valores foram da faixa ASCII, para evitar conflitos com os tokens compostos por um único caractere

Módulo `token`

- Este módulo define uma estrutura para a representação de um token
- Cada token tem um tipo e um atributo associado
- Como o atributo de `NUM` é inteiro e de `ID` é string, foi utilizado o tipo `variant<int, string>` de C++17
- Os tipos dos tokens foram codificados como inteiros, com valores foram da faixa ASCII, para evitar conflitos com os tokens compostos por um único caractere
- Também foi definida uma função para a impressão de um token, que trata internamente as diferenças entre os tipos

Arquivo token.h

```
#ifndef TOKEN_H
#define TOKEN_H

#include <string>
#include <variant>

enum TokenType { NUM = 256, DIV, MOD, ID, DONE };

struct Token {
    int type;
    std::variant<int, std::string> value;

    Token(int t, int v) : type(t), value(v) { }
    Token(int t = DONE, std::string s = "") : type(t), value(s) { }
};

std::ostream& operator<<(std::ostream& os, const Token& token);

#endif
```

Arquivo token.cpp

```
1 #include <ostream>
2 #include "token.h"
3
4 std::ostream&
5 operator<<(std::ostream& os, const Token& token)
6 {
7     switch (token.type) {
8     case NUM:
9         os << "NUM (" << std::get<int>(token.value) << ")";
10        break;
11
12    case ID:
13        os << "ID (" << std::get<std::string>(token.value) << ")";
14        break;
15
16    case DIV:
17        os << "DIV";
18        break;
```

Arquivo token.cpp

```
20      case MOD:
21          os << "MOD";
22          break;
23
24      case DONE:
25          os << "DONE";
26          break;
27
28      default:
29          os << (char) token.type;
30      }
31
32      return os;
33  }
```

Tabela de símbolos

- Este módulo define uma classe para a representação de uma tabela de símbolos

Tabela de símbolos

- Este módulo define uma classe para a representação de uma tabela de símbolos
- Esta classe segue o padrão Singleton, pois deve haver uma única tabela de símbolos, a qual será compartilhada por todas as fases do compilador

Tabela de símbolos

- Este módulo define uma classe para a representação de uma tabela de símbolos
- Esta classe segue o padrão Singleton, pois deve haver uma única tabela de símbolos, a qual será compartilhada por todas as fases do compilador
- A estrutura que armazena os símbolos é um dicionário (classe `map` de C++), onde as chaves são os lexemas e os valores são os tokens

Tabela de símbolos

- Este módulo define uma classe para a representação de uma tabela de símbolos
- Esta classe segue o padrão Singleton, pois deve haver uma única tabela de símbolos, a qual será compartilhada por todas as fases do compilador
- A estrutura que armazena os símbolos é um dicionário (classe `map` de C++), onde as chaves são os lexemas e os valores são os tokens
- Na inicialização da tabela são adicionadas, ao dicionário, todas as palavras reservadas

Tabela de símbolos

- Este módulo define uma classe para a representação de uma tabela de símbolos
- Esta classe segue o padrão Singleton, pois deve haver uma única tabela de símbolos, a qual será compartilhada por todas as fases do compilador
- A estrutura que armazena os símbolos é um dicionário (classe `map` de C++), onde as chaves são os lexemas e os valores são os tokens
- Na inicialização da tabela são adicionadas, ao dicionário, todas as palavras reservadas
- O método `insert()` insere um novo símbolo no dicionário

Tabela de símbolos

- Este módulo define uma classe para a representação de uma tabela de símbolos
- Esta classe segue o padrão Singleton, pois deve haver uma única tabela de símbolos, a qual será compartilhada por todas as fases do compilador
- A estrutura que armazena os símbolos é um dicionário (classe `map` de C++), onde as chaves são os lexemas e os valores são os tokens
- Na inicialização da tabela são adicionadas, ao dicionário, todas as palavras reservadas
- O método `insert()` insere um novo símbolo no dicionário
- O método `find()` localizar um símbolo já inserido, ou retorna vazio, caso o dicionário não tenha nenhum token associado ao lexema passado como parâmetro

Arquivo table.h

```
1  #ifndef SYMBOL_TABLE_H
2  #define SYMBOL_TABLE_H
3
4  #include <bits/stdc++.h>
5  #include "token.h"
6
7  class SymbolTable {
8  public:
9      static SymbolTable& get_instance();
10
11     void insert(const std::string& lexema, const Token& token);
12     std::optional<Token> find(const std::string& lexema) const;
13
14 private:
15     SymbolTable();
16     std::map<std::string, Token> table;
17 };
18
19 #endif
```

Arquivo table.cpp

```
1 #include "table.h"
2
3 SymbolTable& SymbolTable::get_instance()
4 {
5     static SymbolTable instance;
6     return instance;
7 }
8
9 SymbolTable::SymbolTable()
10 {
11     insert("div", Token(DIV));
12     insert("mod", Token(MOD));
13 }
14
15 void
16 SymbolTable::insert(const std::string& lexema, const Token& token)
17 {
18     table[lexema] = token;
19 }
```

Arquivo table.cpp

```
21 std::optional<Token>
22 SymbolTable::find(const std::string& lexema) const
23 {
24     if (table.count(lexema))
25         return table.at(lexema);
26
27     return { };
28 }
```

Módulo scanner

- Este módulo implementa o analisador léxico do tradutor

Módulo `scanner`

- Este módulo implementa o analisador léxico do tradutor
- Como este analisador não tem estado, ele foi implementado por meio de um `namespace`, o que permite usar a mesma notação de método estático de um classe, embora a implementação seja a de uma função regular de C++

Módulo scanner

- Este módulo implementa o analisador léxico do tradutor
- Como este analisador não tem estado, ele foi implementado por meio de um `namespace`, o que permite usar a mesma notação de método estático de um classe, embora a implementação seja a de uma função regular de C++
- A função `next_token()` extrai o próximo token da entrada padrão

Módulo `scanner`

- Este módulo implementa o analisador léxico do tradutor
- Como este analisador não tem estado, ele foi implementado por meio de um `namespace`, o que permite usar a mesma notação de método estático de um classe, embora a implementação seja a de uma função regular de C++
- A função `next_token()` extrai o próximo token da entrada padrão
- O *scanner* ignora todos os espaços em branco

Módulo `scanner`

- Este módulo implementa o analisador léxico do tradutor
- Como este analisador não tem estado, ele foi implementado por meio de um `namespace`, o que permite usar a mesma notação de método estático de um classe, embora a implementação seja a de uma função regular de C++
- A função `next_token()` extrai o próximo token da entrada padrão
- O *scanner* ignora todos os espaços em branco
- Os demais tokens são extraídos conforme a especificação

Arquivo scanner.h

```
1  #ifndef SCANNER_H
2  #define SCANNER_H
3
4  #include "token.h"
5
6  namespace scanner {
7
8      Token next_token();
9
10 };
11
12 #endif
```

Arquivo scanner.cpp

```
1#include <iostream>
2#include "scanner.h"
3#include "table.h"
4
5namespace scanner {
6
7    Token next_token()
8    {
9        auto c = std::cin.get();
10
11        while (isspace(c))
12            c = std::cin.get();
13
14        if (c == EOF)
15            return { DONE, "" };
16
17        if (isdigit(c))
18        {
19            std::cin.unget();
```

Arquivo scanner.cpp

```
21              int value;
22              std::cin >> value;
23
24              return { NUM, value };
25          }
26
27          if (isalpha(c))
28          {
29              std::string lexema;
30
31              while (isalpha(c))
32              {
33                  lexema.push_back(c);
34                  c = std::cin.get();
35              }
36
37              std::cin.unget();
```

Arquivo scanner.cpp

```
39              auto table = SymbolTable::get_instance();
40              auto token = table.find(lexema);
41
42              if (not token)
43              {
44                  table.insert(lexema, Token(ID, lexema));
45                  return { ID, lexema };
46              } else
47                  return token.value();
48          }
49
50          if (isgraph(c))
51              return Token(c);
52
53          return Token(DONE);
54      }
55  }
```

Módulo `parser`

- Este módulo implementa o analisador sintático do tradutor

Módulo `parser`

- Este módulo implementa o analisador sintático do tradutor
- De fato, é um analisador gramatical preditivo que, em conjunto com as ações semânticas especificadas, produz um tradutor de notação infixa para posfixa

Módulo `parser`

- Este módulo implementa o analisador sintático do tradutor
- De fato, é um analisador gramatical preditivo que, em conjunto com as ações semânticas especificadas, produz um tradutor de notação infixa para posfixa
- Ele invoca o *scanner* para obter os tokens da entrada, um por vez

Módulo `parser`

- Este módulo implementa o analisador sintático do tradutor
- De fato, é um analisador gramatical preditivo que, em conjunto com as ações semânticas especificadas, produz um tradutor de notação infixa para posfixa
- Ele invoca o *scanner* para obter os tokens da entrada, um por vez
- Cada não-terminal da gramática é implementado por meio de um procedimento

Módulo `parser`

- Este módulo implementa o analisador sintático do tradutor
- De fato, é um analisador gramatical preditivo que, em conjunto com as ações semânticas especificadas, produz um tradutor de notação infixa para posfixa
- Ele invoca o *scanner* para obter os tokens da entrada, um por vez
- Cada não-terminal da gramática é implementado por meio de um procedimento
- Cada expressão da entrada será traduzida para uma linha da saída, em notação posfixa

Módulo `parser`

- Este módulo implementa o analisador sintático do tradutor
- De fato, é um analisador gramatical preditivo que, em conjunto com as ações semânticas especificadas, produz um tradutor de notação infixa para posfixa
- Ele invoca o *scanner* para obter os tokens da entrada, um por vez
- Cada não-terminal da gramática é implementado por meio de um procedimento
- Cada expressão da entrada será traduzida para uma linha da saída, em notação posfixa
- As expressões da entrada devem ser terminadas por **;**

Arquivo parser.h

```
1  #ifndef PARSER_H
2  #define PARSER_H
3
4  namespace parser {
5
6      void parse();
7
8  };
9
10 #endif
```

Arquivo parser.cpp

```
1#include "scanner.h"
2#include "parser.h"
3#include "table.h"
4#include "error.h"
5
6Token lookahead;
7
8namespace parser {
9
10    void expr();
11    void termo();
12    void fator();
13
14    void reconhecer(const Token& token);
15    void print(const Token& token);
```

Arquivo parser.cpp

```
17    void parse()
18    {
19        lookahead = scanner::next_token();
20
21        while (lookahead.type != DONE)
22        {
23            expr();
24            reconhecer(Token(';'));
25            std::cout << '\n';
26        }
27    }
28
29    void expr()
30    {
31        termo();
```

Arquivo parser.cpp

```
33          while (true)
34          {
35              if (lookahead.type == '+' or lookahead.type == '-')
36              {
37                  auto t = lookahead;
38
39                  reconhecer(lookahead);
40                  termo();
41                  print(t);
42              }
43              else
44                  break;
45          }
46      }
47
48      void termo()
49      {
50          fator();
```

Arquivo parser.cpp

```
52          while (true) {
53              switch (lookahead.type) {
54              case '*':
55              case '/':
56              case DIV:
57              case MOD:
58              {
59                  auto t = lookahead;
60                  reconhecer(lookahead);
61                  fator();
62                  print(t);
63                  break;
64              }
65
66              default:
67                  return;
68              }
69          }
70      }
```

Arquivo parser.cpp

```
72      void fator()
73      {
74          switch (lookahead.type) {
75          case '(':
76              reconhecer(Token('('));
77              expr();
78              reconhecer(Token(')'));
79              break;
80
81          case NUM:
82          case ID:
83              print(lookahead);
84              reconhecer(lookahead);
85              break;
86
87          default:
88              erro("Erro de sintaxe em fator");
89          }
90      }
```

Arquivo parser.cpp

```
92      void reconhecer(const Token& token)
93      {
94          if (token.type == lookahead.type)
95              lookahead = scanner::next_token();
96          else
97              erro("Erro de sintaxe em reconhecer");
98      }
99
100     void print(const Token& token)
101     {
102         switch (token.type) {
103         case '+':
104         case '-':
105         case '/':
106         case '*':
107             std::cout << (char) token.type;
108             break;
```

Arquivo parser.cpp

```
110         case DIV:
111         case MOD:
112             std::cout << token;
113             break;
114 
115         case ID:
116             std::cout << std::get<std::string>(token.value);
117             break;
118 
119         case NUM:
120             std::cout << std::get<int>(token.value);
121             break;
122 
123         default:
124             std::cout << "token desconhecido = " << token << '\n';
125         }
126     }
127 }
```

Módulo `error`

- Este módulo é responsável pelo tratamento de erros

Módulo `error`

- Este módulo é responsável pelo tratamento de erros
- A abordagem utilizada é simplificada: é impressa a mensagem indicada e o programa é encerrado por meio da função `exit()`

Módulo `error`

- Este módulo é responsável pelo tratamento de erros
- A abordagem utilizada é simplificada: é impressa a mensagem indicada e o programa é encerrado por meio da função `exit()`

```
1 #include <iostream>
2 #include "error.h"
3
4 void erro(const std::string& message)
5 {
6     std::cerr << message << '\n';
7     exit(-1);
8 }
```

Referências

1. **AHO**, Alfred V, **SETHI**, Ravi, **ULLMAN**, Jeffrey D. *Compiladores: Princípios, Técnicas e Ferramentas*, LTC Editora, 1995.
2. GNU.org. GNU Bison, acesso em 23/05/2022.
3. Wikipédia. Flex (lexical analyser generator), acesso em 23/05/2022.