Linguagens de Domínio Específico

Fabio Mascarenhas – 2017.1

http://www.dcc.ufrj.br/~fabiom/dsl

Definindo DSLs

- Linguagem específica de domínio: uma linguagem de programação de computadores de expressividade limitada focada em um domínio específico
- Linguagem de programação de computadores: uma linguagem usada por humanos para instruir um computador a fazer algo
- Natureza de linguagem: senso de fluência, composicionalidade
- Expressividade limitada: mínimo de recursos para ser útil ao seu domínio
- Foco no domínio: foco claro em um domínio pequeno

DSLs externas

- Separadas da linguagem principal
- Sintaxe customizada
- Análise sintática usual de linguagens de programação
- Não necessariamente arquivos separados, pode ser embutida em strings na linguagem principal
- Exemplos: expressões regulares, SQL, Graphviz DOT, makefiles, arquivos de configuração

DSLs internas

- Uma maneira estilizada de usar uma linguagem de propósito geral
- Código de uma DSL interna é código válido em sua linguagem principal, mas usa apenas um subconjunto de seus recursos
- Normalmente a linguagem principal tem recursos de *metaprogramação*, seja sintática (*macros*) ou dinâmica (*reflexão*)
- Exemplos: vários frameworks nas linguagens Ruby (especialmente o Ruby on Rails) e Scala, arquivos de configuração em Lua

DSLs internas vs APIs comando-consulta

- Em uma API comando-consulta, seus métodos são de certa forma autossuficientes, e podem ser usados isoladamente

- Em uma DSL interna, seus métodos frequentemente só fazem sentido no contexto de uma expressão que compõe vários desses métodos

```
activeState.addTransition(drawerOpened, waitingForLightState);
```

```
state :active do
  transitions :drawerOpened => :waitingForLight,
              :lightOn => :waitingForDrawer
end
```

```
active
  .transition(drawerOpened).to(waitingForLight)
  .transition(lightOn).to(waitingForDrawer)
  ;
```

DSLs autossuficientes e fragmentárias

- A DSL do controlador da última aula é um exemplo de *DSL autossuficiente*
- Podemos examinar um programa em uma DSL autossuficiente e entender o que ele faz isoladamente do programa principal
- DSLs também podem ser *fragmentária*, usando pedaços de DSL como melhorias em uma linguagem hospedeira
- Dois bons exemplos de DSLs fragmentárias são expressões regulares e SQL

Por que usar uma DSL?

- Aprimorar a produtividade de desenvolvimento – quanto mais fácil de ler um trecho de código e entender o que ele faz, mais fácil é encontrar erros e modifica-lo

- Comunicação com especialistas em domínio – mesmo que um especialista não possa escrever, ele pode ler e entender o que está expresso na DSL

- Mudança no contexto de execução – de tempo de compilação para tempo de execução, ou do programa principal para um sistema de banco de dados

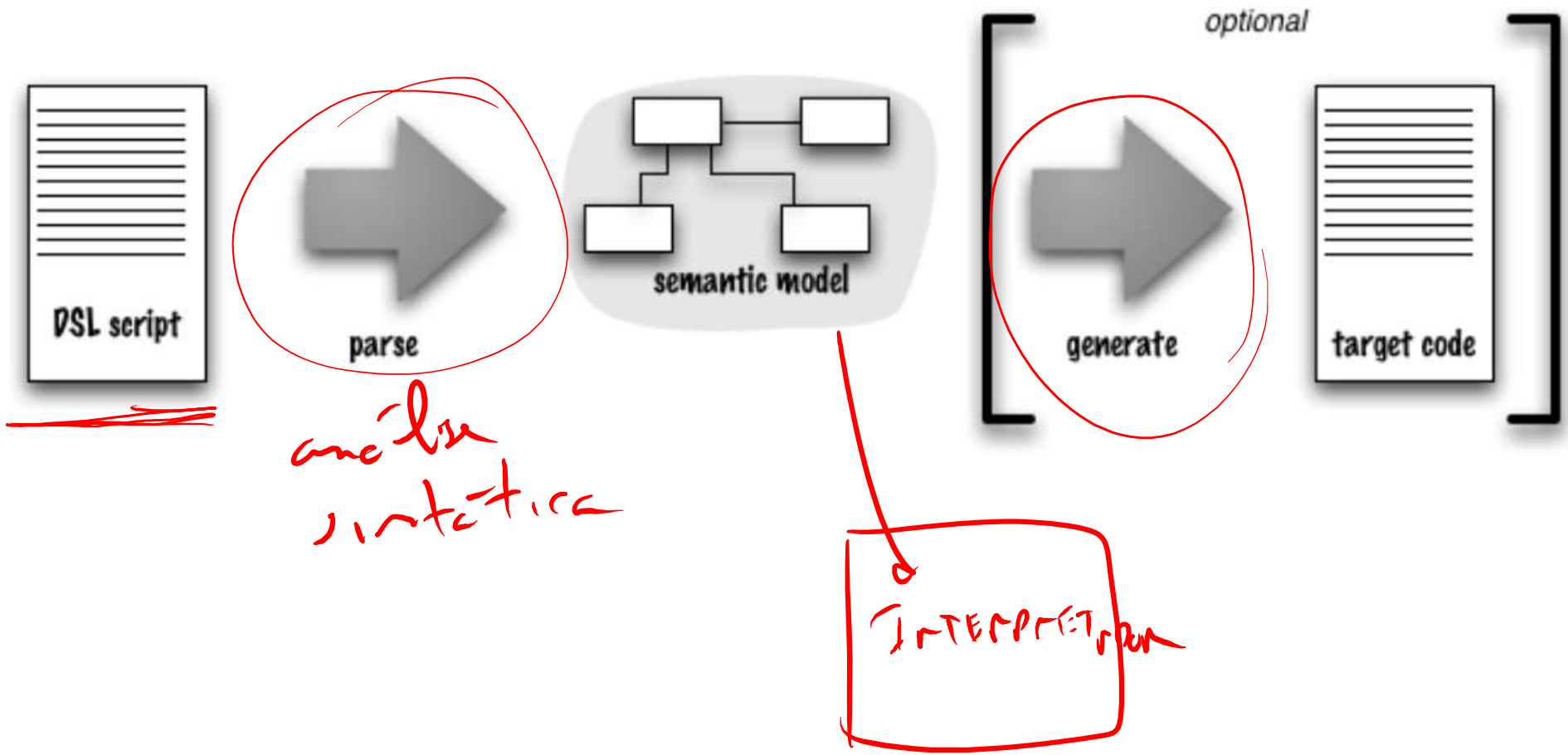
- Modelo computacional alternativo – modelos diferentes do modelo imperativo tradicional podem dar soluções mais simples para diversos problemas

Problemas com DSLs

- Cacofonia de linguagens – ter diversas linguagens no mesmo projeto dificultaria introduzir novas pessoas ao projeto
- Custo de construção – uma DSL tem um custo para ser implementada e mantida, pois requer conhecimento mais especializado
- Feature creep – uma DSL que ganha muitos recursos pode acabar se tornando uma linguagem de propósito geral, usada para desenvolver sistemas completos
- Abstração restrita – a menor expressividade das DSLs pode complicar bastante a solução de problemas que não se encaixam bem no propósito dela

Processamento de uma DSL

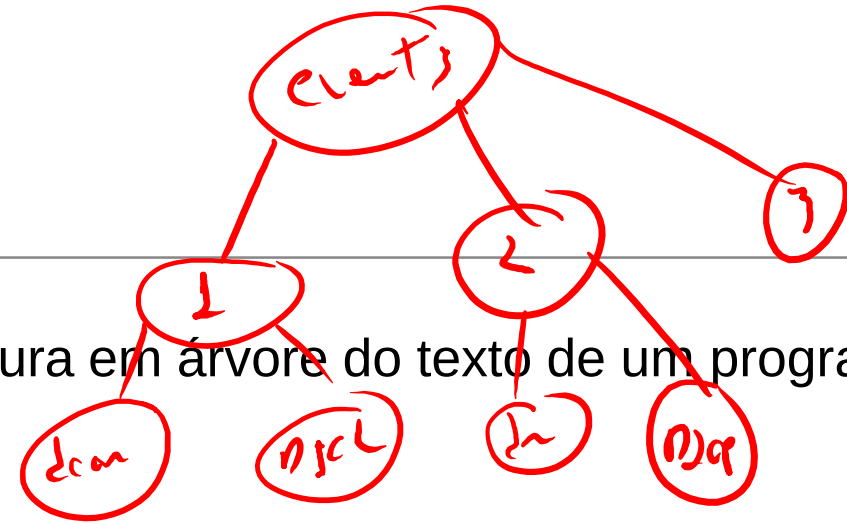
externa e extrutiçiente



Modelo semântico

- Representação abstrata do comportamento dos programas na DSL
- Em uma DSL interpretada, a base para o interpretador
- Pode ser também uma API que a DSL encapsula, nesse caso é independente da DSL
- Podemos até ter várias DSLs e APIs tendo como alvo o mesmo modelo semântico
- Tanto DSLs externas quanto internas usam podem modelos semânticos

Analisador sintático



- O analisador sintático extrai uma estrutura em árvore do texto de um programa

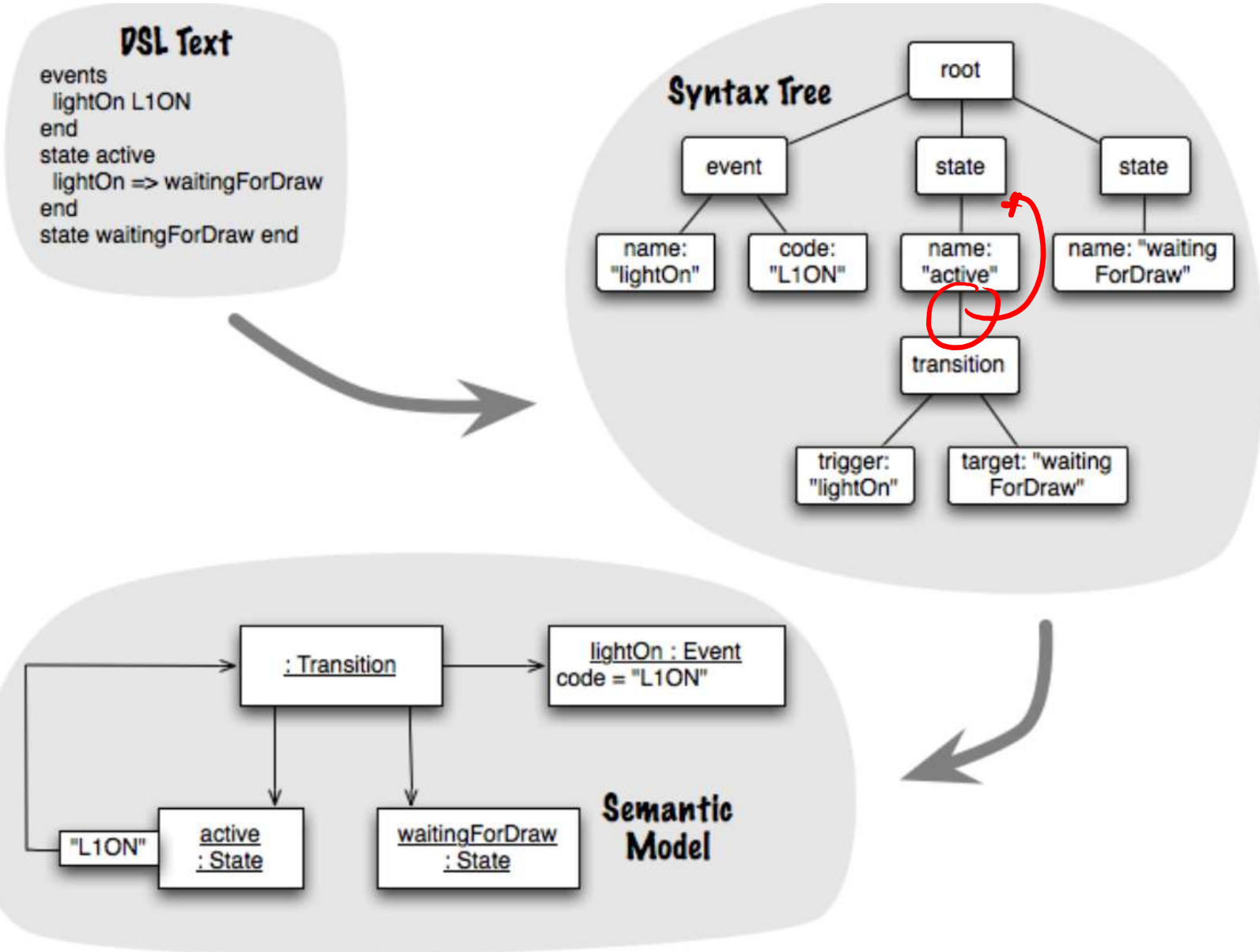
```
events
  doorClosed D1CL
  drawerOpened D2OP
end
```

- Uma estrutura em árvore é uma estrutura hierárquica das partes que compõem o programa

- Não existe uma estrutura única, podemos pensar em várias estruturas diferentes para representar o mesmo programa

- Depois de extraída essa árvore, ela é percorrida para instanciar o modelo semântico (ou o modelo é construído diretamente, e a árvore fica apenas implícita no *call stack* do analisador sintático)

Árvore sintática e modelo semântico



Gramáticas

- Uma gramática é uma descrição formal de como um texto se transforma em uma árvore sintática

- Basicamente é um conjunto de *regras de produção*, onde cada regra possui um termo sintático no lado esquerdo e um modelo de sentença para esse termo no lado direito (formado por outros termos sintáticos)

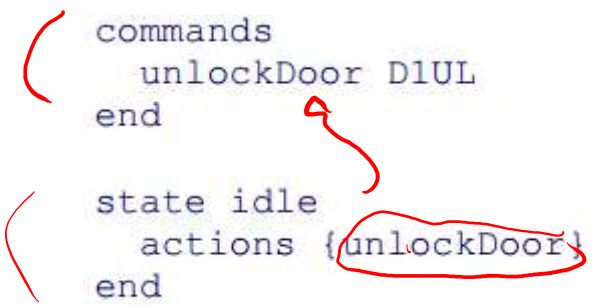
```
additionStatement := number '+' number
```

- Uma linguagem pode ter várias gramáticas!

Contexto

- Depois de construir a árvore, precisamos percorrer a árvore para construir o modelo semântico

- Durante esse percurso, precisamos guardar o *contexto* atual



- No exemplo acima, o nome “unlockDoor” na lista de ações no estado “idle” é uma referência a comando “unlockDoor” definido no bloco de comandos

Exemplo de análise sintática

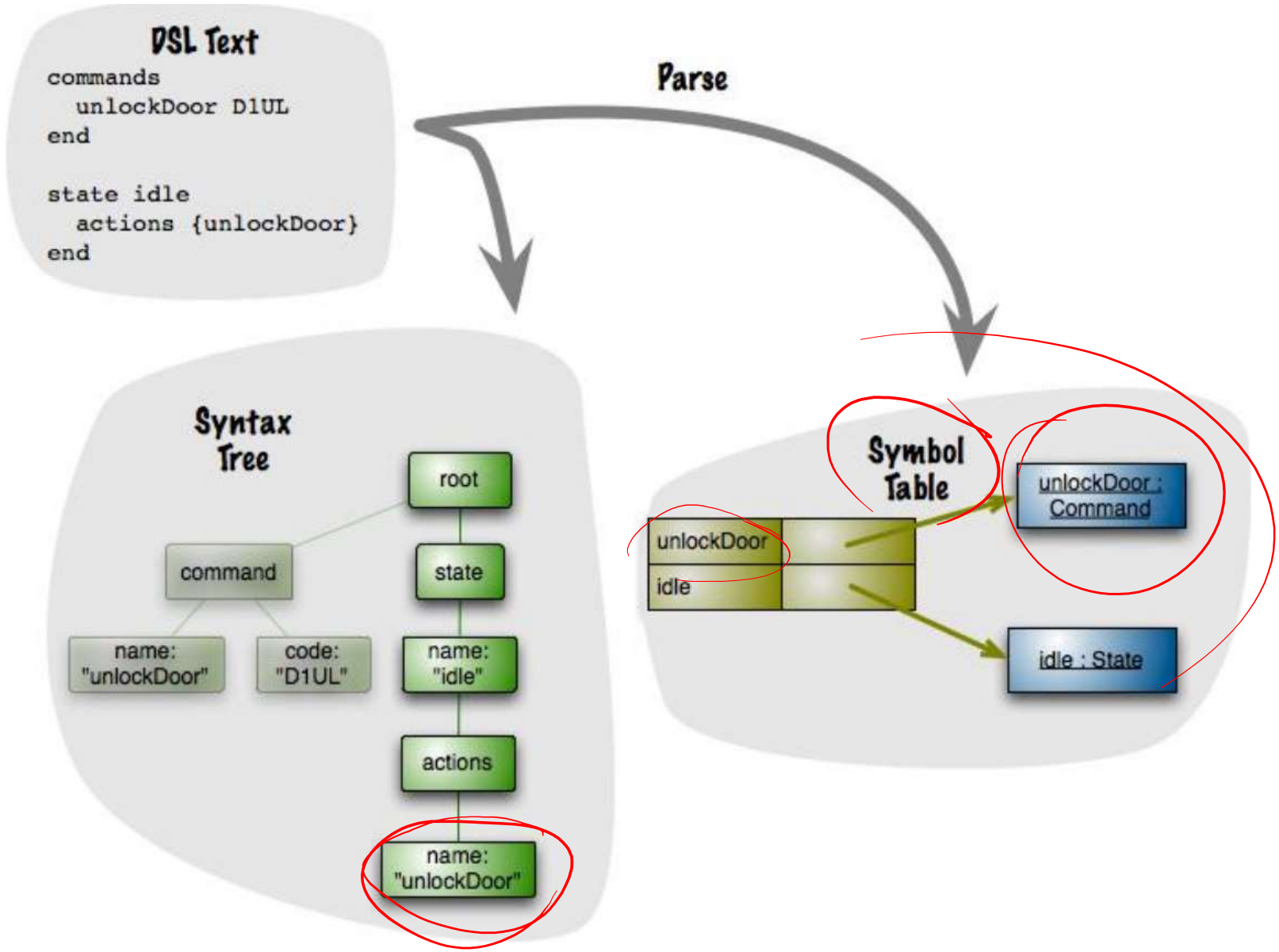


Tabela de símbolos

- Guardamos o contexto em uma *tabela de símbolos*
- Uma tabela de símbolos é um dicionário que associa um nome a um objeto contendo seus atributos
- Definições inserem novos elementos na tabela de símbolos, e referências são consultadas na tabela
- O objeto guardado na tabela depende do tipo de processamento que estamos fazendo da árvore; pode ser um objeto do modelo semântico, por exemplo
- A depender da linguagem mais de um percurso pode ser necessário; por exemplo, a linguagem pode permitir usos antes das definições

Mais contexto

- Outra espécie de contexto aparece quando temos partes de um todo

- Em uma linguagem de programação OO, uma espécie desse tipo de contexto é a classe onde estamos definindo métodos e campos

```
state idle
  actions {unlockDoor}
end

state unlockedPanel
  actions {lockDoor}
end
```

- No exemplo acima, a qual estado as ações pertencem é parte desse contexto

- Podemos representar esse contexto através de *variáveis de contexto*; elas podem inclusive ser armazenadas na tabela de símbolos

Teste de DSLs – modelo semântico

- Podemos dividir os testes de uma DSL nos testes do *modelo semântico,* do *analisador sintático* e dos *programas em si*

- Os testes do modelo semântico se parecem com os testes de unidade tradicionais de uma biblioteca

```
@Test
public void event_causes_transition() {
  State idle = new State("idle");
  StateMachine machine = new StateMachine(idle);
  Event cause = new Event("cause", "EV01");
  State target = new State("target");
  idle.addTransition(cause, target);
  Controller controller = new Controller(machine, new CommandChannel());
  controller.handle("EV01");
  assertEquals(target, controller.getCurrentState());
}
```

Teste de DSLs – analisador sintático

- Podemos testar o analisador sintático com pequenos trechos do código na DSL, conferindo que eles geram o objetos desejados no modelo semântico

```
@Test
public void loads_states_with_transition() {
  String code =
    "events trigger TGGR end " +
    "state idle " +
    "trigger => target " +
    "end " +
    "state target end ";
  StateMachine actual = StateMachineLoader.loadString(code);
  State idle = actual.getState("idle");
  State target = actual.getState("target");
  assertTrue(idle.hasTransition("TGGR"));
  assertEquals(idle.targetState("TGGR"), target);
}
```

Testes negativos

- É importante também testar o analisador com entradas inválidas, e verificar se elas estão gerando erros corretamente

- Podemos ter tanto entradas *sintaticamente inválidas* (algum termo faltando, ou escrito errado) ou *semanticamente inválidas* (uma referência não definida, por exemplo, ou uma operação feita com operandos inválidos)

```
@Test public void targetStateNotDeclared () {
  String code =
    "events trigger TGGR end " +
    "state idle " +
    "trigger => target " +
    "end ";
  try {
    StateMachine actual = StateMachineLoader.loadString(code);
    fail();
  } catch (AssertionError expected) {}
```

Teste de DSLs – testando os programas

- Uma DSL pode não ser expressiva o suficiente para se escrever nela própria
- O implementador da DSL deve fornecer ferramentas em volta do ambiente de execução da DSL que permitam descrever e executar testes
- Esse arcabouço de testes também é útil para testes de integração da implementação da DSL como um tudo
- O arcabouço pode até usar outra DSL!

```
events("doorClosed", "drawerOpened", "lightOn")
        .endsAt("unlockedPanel")
        .sends("unlockPanel", "lockDoor");
```

Tratamento de erros

- Um bom sistema de detecção e comunicação de erros é importante em qualquer linguagem de programação, e DSLs não são exceção
- Um erro deve sempre indicar ao usuário pelo menos uma localização aproximada no programa fonte onde o erro foi detectado, e qual a natureza do erro
- Isso quer dizer que essa informação de localização tem que ser propagada desde o texto até os objetos do modelo semântico
- Outra coisa importante é sempre fornecer uma sintaxe para *comentários* na DSL, assim como uma maneira de obter um *traço* da execução do programa, o que ajuda muito na depuração