

**PDS16inEcplise**

|  |  |
| --- | --- |
|  | André Ramanlal |
|  | Tiago Oliveira |

|  |  |
| --- | --- |
| Orientadores | Tiago Dias |
|  | Pedro Sampaio |

Relatório de progresso realizado no âmbito de Projecto e Seminário,  
do curso de licenciatura em Engenharia Informática e de Computadores  
Semestre de Verão 2015/2016

Abril de 2016

**Instituto Superior de Engenharia de Lisboa**  
Licenciatura em Engenharia Informática e de Computadores

**PDS16inEcplise**

|  |  |
| --- | --- |
| 39204 | André Akshei Manoje Ramanlal |
| 40653 | Tiago José Vital Oliveira |
|  |  |

|  |  |
| --- | --- |
| Orientadores: | Tiago Miguel Braga da Silva Dias |
|  | Pedro Miguel Fernandes Sampaio |

Relatório de progresso realizado no âmbito de Projecto e Seminário,  
do curso de licenciatura em Engenharia Informática e de Computadores  
Semestre de Verão 2015/2016

Abril de 2016

# Resumo

O projeto PDS16inEcplise é uma ferramenta que visa facilitar o desenvolvimento de programas em assembly pds16. A ferramenta em questão é um editor de texto que integra funcionalidades para fazer uma verificação da semântica e da sintaxe em tempo de escrita de código. No geral o projeto está direcionado para a criação de uma extensão que integra o editor de texto para a DSL, este deve ser instalado num Ambiente Integrado de Desenvolvimento (IDE) neste caso o Ecplise [1] ou Intellij [2].

**Palavras-chave:** Ambiente Integrado de Desenvolvimento; Assembly PDS16; Linguagem específica de domínio; Plug-in;

**Índice**

[Resumo iii](#_Toc449656501)

[Directivas, instruções, secções de dados v](#_Toc449656502)

[Lista de Figuras vi](#_Toc449656503)

[Lista de Tabelas viii](#_Toc449656504)

[1. Introdução 1](#_Toc449656505)

[1.1 Enquadramento 1](#_Toc449656506)

[1.2 Motivação 3](#_Toc449656507)

[1.3 Objetivos 3](#_Toc449656508)

[2. PDS16 DSL – Linguagem de Domínio Especifico 6](#_Toc449656509)

[2.1 Entidades da Linguagem 6](#_Toc449656510)

[3. Framework Xtext 8](#_Toc449656511)

[3.1 Regras (Parser Rules) 8](#_Toc449656512)

[3.2 Regras Terminais 9](#_Toc449656513)

[3.3 Validadores 10](#_Toc449656514)

[3.4 Compilador 10](#_Toc449656515)

[4. Progresso do Projeto 12](#_Toc449656516)

[Referências 15](#_Toc449656517)

# Directivas, instruções, secções de dados

# Lista de Figuras

[Figura 1 – Exemplo de um ciclo de desenvolvimento de um programa/aplicação. [1] 1](#_Toc449656490)

[Figura 2 - Código exemplo da definição das regras 8](#_Toc449656491)

[Figura 3 - Código exemplo da definição regras terminais 9](file:///D:\Repositorio\PDS16ASM\Relatorio%20de%20Progresso\Relatorio%20Progresso.docx#_Toc449656492)

[Figura 4 - Código da classe Pds16asmRuntimeModule 9](#_Toc449656493)

[Figura 5 - Exemplo de um validador 10](#_Toc449656494)

# Lista de Tabelas

[Tabela 1 - Diagrama de Gantt relativo à previsão da execução do trabalho. 12](#_Toc449656450)

# Introdução

## 1.1 Enquadramento

No domínio da Informática, um programa consiste no conjunto das instruções que define o algoritmo desenvolvido para resolver um dado problema usando um sistema computacional. Para que esse sistema possa realizar as operações definidas por estas instruções é pois necessário que as mesmas sejam descritas usando a linguagem entendida pela máquina, que consistem num conjunto de bits com valores lógicos diversos. Esta forma de codificação de algoritmos é bastante complexa e morosa, pelo que o processo habitual de desenvolvimento de um programa é feito com um maior nível de abstração, recorrendo a linguagens de programação. A Figura 1 mostra as diferentes fases deste processo quando aplicado ao domínio dos sistemas embebidos, em que as linguagens de programação mais utilizadas são o C e o C++.



Figura 1 – Exemplo de um ciclo de desenvolvimento de um programa/aplicação. [1]

Após a definição do problema e elaboração do algoritmo para a sua solução, o programador começa a escrever o programa numa dada linguagem, resultando assim num ou vários ficheiros fonte. Estes são de seguida compilados através de um compilador ou *assembler*, que primeiramente verificam as regras sintáticas e semântica da linguagem e de seguida geram um ficheiro objeto correspondente a cada ficheiro fonte. O *linker* efetua a ligação entre os diversos ficheiros objeto que compõem o programa e as bibliotecas utilizadas, ficheiros partilháveis que podem conter código, dados e recursos em qualquer combinação. Deste último processo resulta um ficheiro com a descrição do algoritmo codificado pelos programadores em linguagem máquina, i.e. um ficheiro executável. Para garantir a correta implementação da solução desejada, são realizados um conjunto de testes sobre este ficheiro.

Os Ambientes Integrados de Desenvolvimento (*IDEs*) são hoje em dia um enorme apoio no desenvolvimento destes programas, uma vez que não só disponibilizam diversas ferramentas para apoio à produção do código, e.g. um editor de texto, a geração automática de código ou o *refactoring*, como ainda possibilitam a interação com outras ferramentas e aplicações, como é o caso dos compiladores, *linkers*, *debuggers*, controladores de versão, etc.

Recorrendo a estas aplicações, um programador consegue ver a sua produtividade maximizada nas diferentes fases do processo de geração do ficheiro executável correspondente ao seu programa. Por exemplo, a geração automática de código permite poupar bastante tempo na escrita do código fonte do programa, bem como ter o código sempre bem indentado e estruturado. Já a funcionalidade de *syntax highlighting* facilita a leitura e análise do código fonte, para além de potenciar a deteção de erros de sintaxe e/ou de semântica. A utilização de um compilador integrado no IDE também permite acelerar o processo de geração do ficheiro executável, pois evita a saída do editor, a subsequente instanciação do compilador num processo aparte e, caso a compilação seja abortada devido a erros, a procura da linha associada a esse erro novamente no editor com vista à sua correção.

Atualmente, existem IDEs para quase todas as linguagens de programação em uso. Algumas destas aplicações suportam apenas uma linguagem de programação, como por exemplo o Kantharos que apenas suporta PHP ou o DRJava que apenas suporta Java. Não obstante, há vários IDEs no mercado que permitem desenvolver programas e aplicações usando várias linguagens de programação, tais como o Eclipse [1] e o IntelliJ [2] cuja quota de mercado é, à data atual, superior a 80% [3]. Esta versatilidade é normalmente conseguida à custa da adição de *plug-ins* ou *add-ons* específicos para uma dada linguagem de programação ao IDE, que são programas que ajudam adicionar novas funcionalidades aos *plug-ins*. Estes podem ser criados a partir de bibliotecas que dão o suporte à criação dos mesmos.

Apesar da maioria destes IDEs e dos seus *plug-ins* e *add-ons* estarem normalmente associados ao desenvolvimento de programas utilizando linguagens de alto nível, como é o caso do C, C++, C# ou Java, muitas destas aplicações também oferecem suporte à codificação dos programas, ou dos seus módulos, usando linguagens de mais baixo nível, tal como o *assembly* (e.g. o Eclipse).

## 1.2 Motivação

A arquitetura PDS16 [4] foi desenvolvida no Instituto Superior de Engenharia de Lisboa (ISEL), em 2008, com o objetivo de suportar não só uma mais fácil compreensão mas também o ensino experimental dos conceitos básicos subjacentes ao tema “Arquitetura de Computadores”. Esta arquitetura adota a mesma filosofia das máquinas do tipo *Reduced Instruction Set Computer* (RISC), oferecendo o seu ISA (*Instruction Set Architecture*) ao programador 6 registos de uso geral e cerca de 40 instruções distintas, organizadas em três classes: 6 instruções para controlo do fluxo de execução, 18 instruções de processamento de dados e 12 instruções de transferência de dados. O espaço de memória útil, que é partilhado para o armazenamento do código e dos dados dos programas, é endereçável ao byte e tem uma dimensão total de 64 kB.

Atualmente, o desenvolvimento de programas para esta arquitetura pode ser feito utilizando a própria linguagem máquina ou *assembly*. A tradução do código *assembly* para linguagem máquina é realizada recorrendo à aplicação dasm [5], que consiste num *assembler* de linha de comandos que apenas pode ser executado em sistemas compatíveis com o sistema operativo Windows da Microsoft.

Assim, o ciclo de geração de um programa passa por codificá-lo em linguagem *assembly* utilizando um editor de texto simples, tal como o Notepad, e posteriormente invocar a aplicação dasm a partir de uma janela de linha de comandos. Sempre que existam erros no processo de compilação, é necessário voltar ao editor de texto para corrigir a descrição *assembly* do programa e invocar novamente o *assembler*.

## 1.3 Objetivos

Com este trabalho pretende-se implementar um IDE para suportar o desenvolvimento de programas para o processador PDS16 usando a linguagem *assembly* e com as seguintes ferramentas e funcionalidades:

* Um editor de texto que integre ferramentas para fazer uma verificação da semântica e da sintaxe em tempo de escrita de código, de modo a que o programador possa ser alertado de eventuais erros na utilização da linguagem mais cedo e dessa forma otimizar a sua produtividade;
* *Syntax highlighting*,para permitir uma melhor legibilidade do código fonte;
* Integração com um *assembler*, para permitir a compilação dos programas sem necessidade de ter que abandonar o IDE e visualizar no editor de texto os eventuais erros detetados neste processo.

O IDE a desenvolver será baseado na plataforma Eclipse, atendendo à sua maior utilização na produção de programas e aplicações no domínio dos sistemas embebidos [6], onde se insere a utilização da arquitetura PDS16 no ISEL, e no facto dos alunos dos cursos de LEIC e LEETC do ISEL já terem experiência na utilização desta plataforma quando iniciam a frequência da unidade curricular Arquitetura de Computadores.

Para tal, será desenvolvido um *plug-in* para a arquitetura PDS16 utilizado a *framework* Xtext [7], que é uma *framework* genérica para o desenvolvimento de linguagens específicas de domínio (*DSL*). Para além da sua grande atualidade, a *framework* Xtext  apresenta ainda a grande vantagem de, com base numa mesma descrição de uma DSL, permitir gerar automaticamente *plug-ins* também para a plataforma IntelliJ e para vários *browsers.*

# PDS16 DSL – Linguagem de Domínio Especifico

A nossa solução é apresentada neste capítulo. A solução consiste em grandes ideias, desenvolvidas e testadas. Exemplo de indentação do segundo parágrafo.

## 2.1 Entidades da Linguagem

Texto da secção. Seguem-se exemplos de vários parágrafos.

Esta unidade curricular funciona no semestre de Verão de cada ano lectivo. Nos casos de impedimento prolongado justificado (designadamente por doença ou por motivos profissionais no caso dos trabalhadores-estudantes), poderá ser prolongada, havendo lugar à elaboração de outro relatório de progresso e a nova inscrição se o prolongamento for além do período de época especial desse semestre. A entrega da justificação e a sua apreciação deverão ocorrer antes do final do prazo estabelecido para a entrega final.

# Framework Xtext

Xtext é uma framework para o desenvolvimento de linguagem de programação, as chamadas DSL, domain-specific languages. Com o Xtext é possível definir uma linguagem com toda a sua gramatica resultando uma infraestrutura que incluí parser, linker, typechecker, compilador e também a possibilidade de ter um editor através do Ecplise [1], Intellij IDEA [2] e também através do browser.

Esta framework foi desenvolvida com o intuito de ser fácil de aprender e ser possível em poucos minutos descrever uma linguagem simples e também ser possível extrair o projeto em forma de um plug-in para a sua portabilidade entre máquinas.

Esta framework foi usada para a realização de um plug-in para uma linguagem de assembly PDS16. Não havendo nenhum editor de texto para a mesma, comprometemos a usar esta biblioteca.

Para começar a trabalhar tivemos de instalar o plug-in da framework no nosso IDE neste caso o Ecplise. Após criar um novo projeto e definir o nome da linguagem e a sua extensão, começamos a desenvolver. O primeiro passo é definir a sintaxe da linguagem, ou seja definir as regras.

## 3.1 Regras (Parser Rules)

Parser Rules são regras que definem uma sequência de outras regras conjugando com palavras-chaves. Como por exemplo:



Figura 2 - Código exemplo da definição das regras

Statement é uma regra que em que a sua definição é uma das referências para outra regra. Neste caso se virmos a regra Label podemos ver que a sua definição já contem palavras-chaves como “:” e um identificador “labelName” que é o tipo ID considerado um terminal. A regra Ret em que apenas é defenida pela palvra chave ret ou iret. A regra Jump que é mais complexa pode ser defenida por uma destas palavras chaves, seguida pela regra OperationWithOffset.

## 3.2 Regras Terminais

Regra terminal é também uma regra mas só que esta definida por uma sequência de caracteres também chamadas por tokens rules ou lexer rules. Um terminal pode retornar um tipo, por definição eles retornam sobre a forma de String: ecore::EString. Mas é possível converter o tipo de retorno para um típico especifico desque seja uma instancia de ecore::EDataType. Para isso é necessário implementar a interface IValueConverter e criar o respetivo converter de String para o tipo pretendido.

Figura 3 - Código exemplo da definição regras terminais

O primeiro terminal, ID, começa com um caracter de ‘a’ a ‘z’ ou por um ‘\_’ seguindo de nenhum ou mais caracteres incluindo números. O terminal HEX é a definição de um número hexadecimal, mas retornando um número inteiro em vez da String. Para que isso fosse possível foi necessário acrescentar um método a classe “Pds16RunTimeModule” o código seguinte da figura 4.



Figura 4 - Código da classe Pds16asmRuntimeModule

Este método apresentado na classe, retorna uma instância de “Pds16asmValueConverter” que por sua vez retorna uma instância de um “ValueConverter” específico dependendo do tipo do terminal, que têm a função para converter de String para inteiro.

## 3.3 Validadores

Existem certas regras de uma linguagem que não podem ser definidas, logo essas tem que ser verificadas no ato da compilação. Mas tal como um editor de texto o Xtext permite que sejam feitas essas verificações ao decorrer da escrita do código indicando o erro. Os validadores da framework permitem analisar determinado conteúdo e indicar ao utilizador do erro, retirando essa função ao compilador, pois não é possível compilar com erros de validações. No caso do nosso no trabalho verificamos os limites dos números conforme o tipo, por exemplo o offset 8 que só pode estar compreendido entre minimio valor a 8 bits com sinal e o máximo valor a 8 bits com sinal. A figura 5 mostra o código que permite validar o valor.



Figura 5 - Exemplo de um validador

## 3.4 Compilador

A própria framework disponibiliza a opção de criar um compilador, mas nesta etapa do projeto decidimos usar um compilado externo, o DASM [5]. Para isso, é feita uma chamada ao compilador externo através do “ProcessBuilder”, passando como input o ficheiro fonte, asm, capturando o output após a compilação. Com esse output analisamos e conseguimos determinar se o ficheiro fonte tem erros. Caso tenha erros, assinalamos no ficheiro fonte com os erros na respetiva linha e com a mensagem que o compilador deu.

# Progresso do Projeto



Tabela 1 - Diagrama de Gantt relativo à previsão da execução do trabalho.

No período decorrido da realização deste projeto (7 semanas) conseguimos alcançar os seguintes objetivos:

* **Estudo do Assembly PDS16:** Estudo da linguagem com base na documentação de Arquitetura de Computadores Capitulo 13 [1] e 15 [2].
* **Estudo da Framework Xtext:** Estudo da framework com base na documentação disponibilizada na Web.
* **Elaboração Proposta do Projeto:** Foi elaborada a proposta do projeto depois do estudo tanto do assembly pds16 e da framework que vai servir de suporte ao projeto, tendo feito uma calendarização com prazos a cumprir.
* **Implementação do ASM PDS16:** Foi definida a sintaxe gramatical da linguagem utilizando a framework Xtext [3], criando também validadores para certos aspetos da linguagem que ajudam ao utilizador informando os erros.
* **Gerador (Utilizando PDS16):** Para compilar o ficheiro foi chamado o compilador DASM passando como input o ficheiro fonte, e recebendo como output o resultado da compilação.Com o output verificamos se existiam erros de compilação, e caso existisse assinalávamos no ficheiro fonte com a mensagem de erro do compilador.

Em termos da calendarização podemos concluir que não estamos atrasados apesar de termos ultrapassado alguns dias em alguns dos pontos referidos acima, conseguindo recuperar até ao momento atual. Como não estamos atrasados em relação ao projeto e prevemos cumprir a calendarização da proposta do projeto.

# Referências

|  |  |
| --- | --- |
| [1] | J. Paraíso, “PDS16. Arquitetura de Computadores – Textos de apoio às aulas teóricas (págs. 13-1 – 13-27),” Lisboa, 2011. |
| [2] | J. Paraíso, “Desenvolvimento de Aplicações. Arquitetura de Computadores – Textos de apoio às aulas teóricas (págs. 15-2 – 15-5),” Lisboa, 2011. |
| [3] | “Xtext 2.5 Documentation, Eclipse Foundation,” 2013. [Online]. Available: http://www.eclipse.org/Xtext/documentation/2.5.0/Xtext%20Documentation.pdf. [Acedido em 05 02 2016]. |
| [4] | T. Dias, “Elaboração de Ficheiros Executáveis,” 2013. [Online]. Available: https://adeetc.thothapp.com/classes/SE1/1314i/LI51D-LT51D-MI1D/resources/2334. [Acedido em 27 03 2016]. |
| [5] | “IDE Ecplise,” [Online]. Available: http://www.eclipse.org. |
| [6] | “Intellij, IDE,” [Online]. Available: https://www.jetbrains.com/idea/. |
| [7] | O. White, “IDEs vs. Build Tools: How Eclipse, IntelliJ IDEA & NetBeans users work with Maven, Ant, SBT & Gradle,” 2014. [Online]. Available: http://zeroturnaround.com/rebellabs/ides-vs-build-tools-how-eclipse-intellij-idea-netbeans-users-work-with-maven-ant-sbt-gradle/. [Acedido em 25 03 2016]. |
| [8] | C. Ajluni, “Eclipse Takes a Stand for Embedded Systems Developers,” [Online]. Available: http://www.embeddedintel.com/search\_results.php?article=142. [Acedido em 30 03 2016]. |