



CENTRO DE EDUCAÇÃO SUPERIOR DE BRASÍLIA
INSTITUTO DE EDUCAÇÃO DE BRASÍLIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

BÁRBARA ALVES BEZERRA DOS ANJOS
WANDERLAN SANTOS DOS ANJOS

LLVM-PASCAL
Um compilador Object Pascal usando LLVM

Brasília-DF
Junho, 2012

BÁRBARA ALVES BEZERRA DOS ANJOS
WANDERLAN SANTOS DOS ANJOS

LLVM-PASCAL

Um compilador Object Pascal usando LLVM

Trabalho de Conclusão de Curso apresentado ao Curso de
Ciência da Computação do Instituto de Educação Superior
de Brasília, como requisito parcial para obtenção do grau
de Bacharel em Ciência da Computação.

Orientador: Prof. PhD Joel Guilherme da Silva Filho

Brasília-DF
Junho, 2012

BÁRBARA ALVES BEZERRA DOS ANJOS
WANDERLAN SANTOS DOS ANJOS

LLVM-PASCAL

Um compilador Object Pascal usando LLVM

Trabalho de Conclusão de Curso aprovado pela
Banca Examinadora com vistas a obtenção do título
de Bacharel em Ciência da Computação do Instituto
de Educação Superior de Brasília.

Brasília, DF 08 de junho de 2012.

Banca Examinadora:

Prof. PhD Joel Guilherme da Silva Filho
Orientador

Prof. MSc. Paulo Guilherme Teixeira Freire

Profª. MSc. Flávia Maria Alves Lopes

Para todas as pessoas que gostam de programar em
Object Pascal e gostam de inovações tecnológicas.
Cujo desejo de inovação impregnou cada página
deste trabalho.

AGRADECIMENTOS

Em primeiro lugar agradecemos ao nosso querido Jesus, por encher as nossas vidas de fé, esperança e amor, esses três. Mas o maior desses é o AMOR.

Gostaríamos de agradecer às nossas famílias pelo incentivo aos estudos e pelo apoio em todos os momentos da formação educacional.

Agradecemos às pessoas que acessaram o site do nosso projeto: <http://llvm-pascal.googlecode.com> que, mandaram e-mails com ideias que auxiliaram na construção do nosso projeto. Em especial a Aleksey Naumov de Surgut, Rússia Central, que colabora com o projeto e criou a interface do LLVM em Object Pascal.

Agradecemos aos nossos orientadores Prof. PhD Joel Guilherme da Silva Filho e Prof. MSc. Paulo Guilherme Teixeira Freire, pelo apoio e transmissão de conhecimentos no campus do Centro Universitário IESB – Instituto de Educação Superior de Brasília.

Agradecemos a todos os amigos que tivemos a oportunidade de conhecer no campus do Centro Universitário IESB – Instituto de Educação Superior de Brasília.

Por último, com todo carinho e admiração, gostaríamos de agradecer ao corpo docente do Departamento de Ciência da Computação do Centro Universitário IESB - Instituto de Educação Superior de Brasília. especialmente à Profa. MSc. Patrícia Moscariello Rodrigues, pela paciência e amor com que conduziu a nossa história estudantil nos momentos mais difíceis.

Nós não teríamos conseguido concluir esse curso de graduação se não fosse pela cordial ajuda e compreensão de todas essas pessoas. Obrigado a todos.

*“Se eu fui capaz de ver mais longe é porque estava
de pé nos ombros de gigantes.”*

*Isaac Newton - Carta para Robert Hooke
(15 de Fevereiro de 1676)*

RESUMO

Este trabalho apresenta a construção de um compilador que pode ser usado para desenvolvimento de aplicações reais, mas que ao mesmo tempo pode ter seu código fonte usado para fins didáticos, servindo de estudo de caso para técnicas de compilação modernas. A linguagem escolhida foi o Object Pascal, porque há uma grande base de aplicações, bibliotecas, utilitários e componentes, muitos com fonte aberto. Em 2005 havia mais de 1,7 milhão de programadores Object Pascal. O dialeto Object Pascal que se tornou o padrão de fato de mercado foi o Object Pascal usado no Delphi. Apenas dois compiladores atualmente podem compilar tais aplicações, um comercial (Delphi, fonte fechado que até a versão XE2 é desenvolvido em assembly x86 e ANSI-C) e outro *open source* (Free Pascal, desenvolvido em Pascal). Considerando o Free Pascal, por que então criar outro compilador Object Pascal open source? Por causa do tamanho e da complexidade do compilador Free Pascal, que ao longo de mais de 15 anos de desenvolvimento e contribuições de 55 desenvolvedores ativos, tem hoje mais de 255.000 linhas de código (OHLOH, 2012), o que o torna inviável como material de estudo e difícil de se entender mesmo para programadores Object Pascal experientes. A proposta do LLVM-Pascal é ter uma funcionalidade similar com muito menos linhas de código. Para tanto o compilador está dividido em um *frontend* escrito em Object Pascal e em um *backend* de mercado escrito em C++, o LLVM da Apple. A ideia é que o LLVM-Pascal seja para o Free Pascal/Delphi, o que o [Minix](#) é para o Linux/Unix, e ainda possa ser usado nos mesmos projetos que utilizam o Free Pascal e o Delphi.

Palavras-chave: Compilador. LLVM. Object Pascal. Free Pascal. *Frontend*. *Backend*. C++. LLVM-Pascal. Delphi. *Open source*. Compilação.

ABSTRACT

This paper presents the construction of a compiler that can be used for the development of real applications, but at the same time may have its source code used for teaching purposes, serving as a case for modern compiling techniques. The chosen language is Object Pascal, for which there is a large base of applications, libraries, utilities and components, many with open source. In 2005 there were more than 1.7 million Object Pascal programmers. The Object Pascal dialect that became the de facto market standard was Object Pascal used in Delphi. Only two compilers currently can build such applications, a commercial (Delphi, closed source until version XE2, which is developed in x86 assembly and ANSI-C) and another open source (Free Pascal, developed in Pascal). Considering the Free Pascal, so why to create another open source Object Pascal compiler? The big motivator is the size and complexity of the Free Pascal compiler which, for over 15 years of development and contributions from 55 active developers, today has more than 255,000 lines of code (OHLOH, 2012), what makes it impractical to study and difficult to understand, even for experienced Object Pascal programmers. The purpose of LLVM-Pascal is having a similar functionality with far fewer lines of code. For this purpose the compiler is divided in two modules: a frontend, written in Object Pascal, and a backend "state of the art", written in C++, the Apple LLVM. The idea is that the LLVM-Pascal can be to Free Pascal/Delphi, the same that Minix is to Linux/Unix, and still be able to be used in projects where Free Pascal and Delphi can.

Keywords: Compiler. LLVM. Object Pascal. Free Pascal. Frontend. Backend. C++. Delphi. LLVM-Pascal. Open source. Compiling.

LISTA DE FIGURAS

Figura 1 – Visão geral de um compilador.....	11
Figura 2 – Os três dragões da literatura de compiladores.....	12
Figura 3 - As duas fases principais de um compilador.....	15
Figura 4 - Fases do compilador Object Pascal usando LLVM.....	16
Figura 5 – Sistema de desenvolvimento com LLVM-Pascal.....	17
Figura 6 – O frontend Object Pascal sendo redirecionado	18
Figura 7 – Página inicial do site do LLVM-Pascal no GoogleCode.....	19
Figura 8 – Bloco de um T-diagram.....	20
Figura 9 – Bootstrapping do LLVM-Pascal, usando Free Pascal.....	20
Figura 10 – Bootstrapping do backend LLVM, usando GCC	21
Figura 11 - O novo compilador LLVM-Pascal após o bootstrapping.....	21
Figura 12 – Self-hosting do compilador migrando de uma versão N para N+1.....	21
Figura 13 – As bibliotecas RTL, FCL e LCL e seus níveis de abstração.....	26
Figura 14 – Exemplo de programa Pascal simples.....	27
Figura 15 – Saída do exemplo do programa simples.....	27
Figura 16 - Programa Alô Mundo.....	36
Figura 17 - Programa Alô Mundo executando.....	37
Figura 18 – Exemplo com Writeln.....	38
Figura 19 - Resultado do programa da Figura 18.....	38
Figura 20 – Saída formatada corretamente.....	39
Figura 21 - Utilização de Readln.....	40
Figura 22 - Soma de ponto flutuante.....	40
Figura 23 - Interface entre o analisador léxico e o sintático.....	41
Figura 24 – Exemplo de mensagem de erro.....	42
Figura 25 – Uma macro em Object Pascal.....	42
Figura 26 – Classe TToken.....	43
Figura 27 – Regex para reconhecer um identificador e o código correspondente.....	45
Figura 28 - Interação do analisador sintático com as demais fases do compilador.....	46
Figura 29 – BNF do parágrafo anterior.....	47
Figura 30 – Análise descendente para um programa Hello World.....	49
Figura 31 – Gramática de exemplo para FIRST(<Start>).....	51

Figura 32 – BNF transformada em SDT refatorada com terminais FIRST.....	52
Figura 33 – Algoritmo da máquina virtual de pilha.....	54
Figura 34 – Máquina virtual de pilha alimentada pela gramática.....	54
Figura 35 – Escopo estático em um programa Pascal.....	56
Figura 36 – Configuração da Tabela de Símbolos ao aninhar o Escopo2 no Escopo1.....	58
Figura 37 – Código de três endereços para arrays.....	59
Figura 38 – Instruções de controle de fluxo para código de três endereços.....	60
Figura 39 – Leiaute do código para o comando if.....	60
Figura 40 - Tradução de uma expressão usando instrução de três endereços.....	61
Figura 41 - Expressões usando arrays.....	61
Figura 42 – Exemplo mais completo com arrays.....	61
Figura 43 – BNF para expressão aritmética.....	62
Figura 44 – BNF para expressão aritmética com ações semânticas.....	63
Figura 45 – Algoritmo: Forma infixada para notação polonesa.....	64
Figura 46 – Fonte do micro compilador.....	68
Figura 47 - Exemplo de compilação do micro compilador.....	69
Figura 48 - Código intermediário gerado pelo micro compilador.....	69

LISTA DE TABELAS

Tabela 1 – Tipos Ordinais.....	30
Tabela 2 – Tipos Reais.....	32
Tabela 3 – Tipos String.....	33
Tabela 4 - Operadores aritméticos.....	34
Tabela 5 - Operadores lógicos.....	34
Tabela 6 - Operadores relacionais.....	35
Tabela 7 - Precedência dos operadores para o micro compilador.....	64

LISTA DE SIGLAS

.bc	Extensão dos arquivos contendo <i>bitcode</i> , também chamado LLVM-IR
.exe	Extensão dos programas executáveis na plataforma Windows
.lpr	Lazarus Project. Extensão para o programa principal Pascal no IDE Lazarus
.obj	Extensão dos arquivos objeto na plataforma Windows
.pas	Extensão dos arquivos-fonte em Object Pascal
ALGOL	ALGOrithmic Language, linguagem que deu origem ao Pascal
AMD	Advanced Micro Devices, fabricante de microprocessadores
ANSI	American National Standards Institute
ARM	Advanced Risc Machine, uma arquitetura usada em sistemas embarcados
AST	Abstract Syntax Tree
BNF	Backus-Naur Form
BSD	Licença estilo Berkeley Software Distribution, mais permissiva que LGPL
CellSPU	Cell Synergistic Processor Unit, processador usado no PlayStation 3
CISC	Complex Instruction Set Computer
Clang	C Language frontend, <i>frontend</i> das linguagens C/C++/Objective C para LLVM
DOS	Disk Operating System
FCL	Free Component Library
Fortran	FORmula TRANslation, primeira linguagem para computação científica
GCC	GNU “C” Compiler ou GNU Compiler Collection
GDB	GNU DeBugger
GLC	Gramática Livre de Contexto
GNU	Gnu is Not Unix
GPL	General Public License
GUI	Graphic User Interface
IDE	Integrated Development Environment
IESB	Instituto de Educação Superior de Brasília
Intel	INTEgrated ELEtrronics, fabricante de microprocessadores
IR	Intermediate Representation ou <i>bitcode</i>
iOS	Internet enabled Operating System, sistema operacional da Apple para dispositivos móveis
JIT	Just In Time compiling

KISS	Keep It Short and Simple
kloc	Kilo Lines Of Code (milhares de linhas de código)
Mac OS X	Macintosh Operating System based on uniX
LCL	Lazarus Component Library
LEX	LEXical analyzer generator
LGPL	Lesser General Public License, mais permissiva que GPL
LLVM	Low Level Virtual Machine
LLVM-IR	Low Level Virtual Machine – Intermediate Representation ou <i>bitcode</i>
loc	Lines Of Code (linhas de código)
Minix	Mini Unix, um Unix baseado em arquitetura microkernel
MIPS	Microprocessor without Interlocked Pipeline Stages, uma arquitetura RISC
PowerPC	Performance Optimization With Enhanced Risc – Performance Computing, arquitetura RISC da IBM
OS/2	Operating System 2, sistema operacional da IBM para PCs
RAD	Rapid Application Development
Regex	Regular Expression
RISC	Reduced Instruction Set Computer
SDT	Syntax Directed Translation
SDTF	Syntax Directed Translation with FIRST terminals
SPARC	Scalable Processor ARChitecture, arquitetura RISC da Sun/Oracle
RTL	Run Time Library
SSA	Static Single Assignment
ST	Symbol Table
TCC	Trabalho de Conclusão de Curso
TG	Trabalho de Graduação
Unix	UNiplexed Information and Computing Service, Unics -> Unix
VCL	Visual Component Library
WHIRL	Winning Hierarchical Intermediate Representation Language, IR do Open64
Win32	Sistema Operacional Windows de 32 bits
Win64	Sistema Operacional Windows de 64 bits
WinCE	Windows Compact Embedded ou Windows Mobile, sistema operacional da Microsoft para dispositivos móveis
X86	Arquitetura Intel CISC de 32 bits
YACC	Yet Another Compiler to Compiler

SUMÁRIO

1 INTRODUÇÃO.....	11
1.1 A IMPORTÂNCIA DOS COMPILADORES.....	11
1.2 OBJETIVOS	13
1.3 JUSTIFICATIVA.....	14
1.4 ESCOPO DO TRABALHO.....	15
1.5 AMBIENTE DE DESENVOLVIMENTO.....	17
1.6 DIFERENCIAL DO PROJETO.....	18
1.7 DESENVOLVIMENTO COLABORATIVO	18
1.8 METODOLOGIA DE DESENVOLVIMENTO.....	19
2 OBJECT PASCAL	22
2.1 BORLAND PASCAL.....	23
2.2 CARACTERÍSTICAS.....	23
2.3 IMPLEMENTAÇÃO.....	24
2.4 RTL, FCL E LCL.....	25
2.5 UM EXEMPLO DE PROGRAMA EM OBJECT PASCAL.....	26
2.6 CARACTERÍSTICAS GERAIS.....	27
2.7 ELEMENTOS DA LINGUAGEM.....	27
2.8 COMANDOS.....	33
2.9 ESTRUTURA BÁSICA DE UM PROGRAMA.....	35
2.10 ESCRIVENDO DADOS NO TERMINAL.....	37
2.11 ENTRADA DE DADOS.....	39
3 ANALISADOR LÉXICO.....	41
3.1 SEPARAÇÃO DO ANALISADOR LÉXICO DO SINTÁTICO.....	42
3.2 O TOKEN	43
3.3 EXPRESSÕES REGULARES	44
4 ANALISADOR SINTÁTICO.....	46
4.1 GRAMÁTICA.....	47

4.2	COMPLEXIDADE.....	48
4.3	TIPOS DE ANALISADORES SINTÁTICOS.....	48
4.4	TERMINAIS FIRST/FOLLOW.....	50
4.5	RECUPERAÇÃO DE ERROS COM FIRST	51
4.6	SDT - SYNTAX DIRECTED TRANSLATION.....	52
4.7	MÁQUINA VIRTUAL DE PILHA.....	53
5	ANALISADOR SEMÂNTICO.....	55
5.1	TABELAS DE SÍMBOLOS.....	56
5.2	DIFERENCIAL DA TABELA DE SÍMBOLOS NO LLVM-PASCAL.....	57
6	GERADOR DE CÓDIGO INTERMEDIÁRIO.....	59
6.1	CÓDIGO DE TRÊS ENDEREÇOS.....	59
6.2	TRADUÇÃO DE COMANDOS.....	60
6.3	TRADUÇÃO DE EXPRESSÕES.....	61
7	MICRO COMPILADOR DE EXPRESSÃO ARITMÉTICA.....	62
7.1	A LINGUAGEM FONTE.....	62
7.2	ANÁLISE LÉXICA.....	62
7.3	ANÁLISE SINTÁTICA.....	62
7.4	ANÁLISE SEMÂNTICA E GERAÇÃO DE CÓDIGO.....	63
7.5	O MICRO COMPILADOR PARA EXPRESSÃO ARITMÉTICA.....	64
8	BACKEND LLVM.....	70
8.1	LLVM – IR.....	71
9	CONCLUSÃO.....	72
9.1	TRABALHOS FUTUROS.....	72
	REFERÊNCIAS BIBLIOGRÁFICAS.....	74
	ANEXO A– GRAMÁTICA DO OBJECT PASCAL.....	77

1 INTRODUÇÃO

Este TCC discute a construção do primeiro compilador para a linguagem Object Pascal usando um *backend* de mercado, o LLVM (Low Level Virtual Machine), que é disponibilizado através de uma licença *open source*.

Este compilador lê e compila fontes da linguagem Object Pascal de acordo com as especificações dos guias de referência do Free Pascal (CANNEYT, 2012) e do Delphi (EMBARCADERO, 2012) que descrevem o Object Pascal adotado pelo mercado.

O LLVM-Pascal é capaz de gerar executáveis para as arquiteturas suportadas pelo LLVM, quatorze até a versão 3.1 (LLVM, 2012). Ele usa as bibliotecas disponibilizadas pelo Free Pascal, licenciadas como *open source*, que permitem que seus executáveis rodem em treze sistemas operacionais diferentes.

O objetivo principal é criar um compilador simples, inteligível, facilmente gerenciável, mais amigável para programadores, mostrando mensagens de erros mais efetivas e claras, facilitando seu uso tanto por aqueles que estão aprendendo a linguagem, quanto por profissionais que queiram usar o compilador em trabalhos e projetos.

1.1 A IMPORTÂNCIA DOS COMPILADORES

As linguagens de programação estão presentes de forma direta nas máquinas que são usadas hoje em dia. Com linguagens de programação são criadas tecnologias e inovações para facilitar a vida das pessoas, simplesmente mudando o comportamento das máquinas. O termo “linguagem de programação” se refere a uma notação textual, fácil de ser entendida por pessoas, que descreve as tarefas a serem executadas por um computador. Mas para que as máquinas interpretem as linguagens de programação é necessário um processo especial de tradução, chamado compilação, que transforma a notação textual para instruções de máquina que um computador pode executar diretamente (AHO, 2008). O compilador é o programa de computador capaz de realizar tais compilações, conforme mostra a Figura 1.



Figura 1 – Visão geral de um compilador

Um compilador é um programa complexo que pode ter milhares ou milhões de linhas de código. Escrever, entender e manter compiladores é uma tarefa árdua e desafiadora que, desde muito tempo, vem sendo comparada jocosamente à luta entre um cavaleiro medieval e um gigantesco dragão cuspidor de fogo (AHO et al., 1977/1986/2008, capa). Uma metáfora que representa o desafio de se conquistar o domínio da complexidade de um compilador, através do conhecimento de diversas disciplinas e áreas de pesquisa, ver Figura 2.

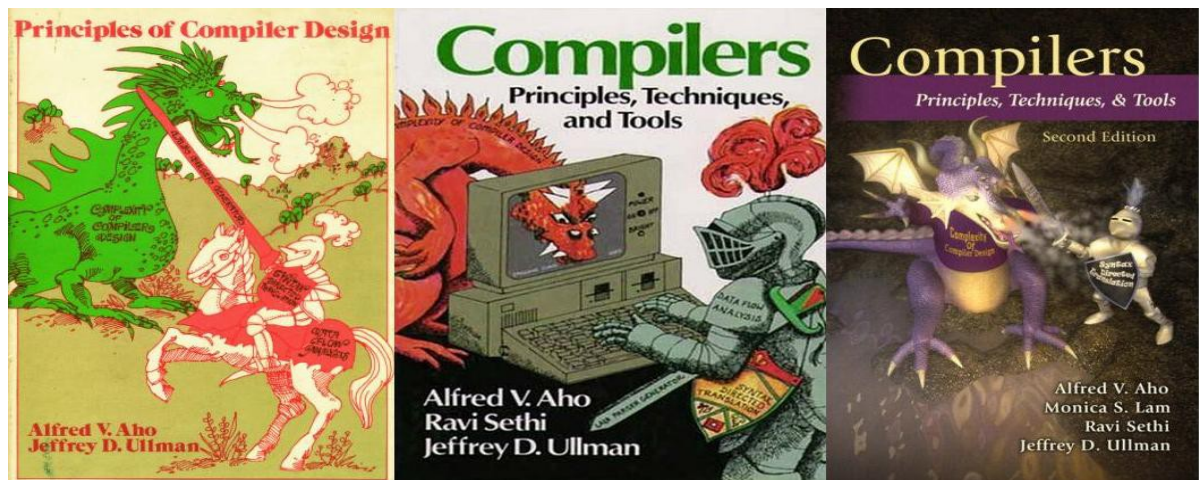


Figura 2 – Os três dragões da literatura de compiladores
green dragon book, red dragon book e purple dragon book

Fonte: Amazon (AMAZON, 2012)

Tais disciplinas são aplicáveis em qualquer forma de computação, o que as torna, particularmente, úteis para qualquer profissional ligado à área da computação. Entre tais disciplinas e áreas de pesquisa destacam-se:

- a) Arquitetura de computadores;
- b) Paradigmas de linguagens de programação;
- c) Construção e análise de algoritmos;
- d) Estruturas, busca e ordenação de dados;
- e) Linguagens formais e autômatos;
- f) Gramáticas livres de contexto;
- g) Teoria da computação;
- h) Programação orientada a objetos;
- i) Sistemas operacionais;

j) Programação concorrente e

k) Engenharia de software.

Ou seja, disciplinas que compõem a espinha dorsal da maioria dos cursos de Ciência da Computação (IESB, 2012). Mas que na construção de um compilador precisam ser usadas de forma combinada, ordenada e sinérgica.

1.2 OBJETIVOS

O objetivo principal deste trabalho é vencer o desafio proposto pela metáfora do dragão, que é criar um compilador simples, inteligível, facilmente gerenciável, que possa ser usado em aplicações científicas e comerciais do mundo real e ainda como material de estudo para fins didáticos.

Certamente um objetivo ambicioso, mas alcançável, se for executado por etapas, escolhendo e combinando as melhores técnicas propostas pela literatura de compiladores e levando em consideração dois princípios norteadores:

- a) Pragmatismo: Usar o máximo de componentes *open source*, prontos, reutilizáveis e ativamente desenvolvidos, que possam ser conectados para obtenção do compilador final, com o menor esforço possível. Parafraseando Isaac Newton (NEWTON, 1676): “Suba nos ombros dos gigantes para ver mais longe”. Por isso o LLVM da Apple, um *framework* de mercado, foi escolhido para construir o *backend* do compilador.
- b) Minimalismo ou KISS (Keep It Short and Simple): Escolher e aperfeiçoar as técnicas de maneira a torná-las tão simples e gerenciáveis em linhas de código quanto possível. Buscar o minimalismo, sempre que possível, sem prejuízo da corretude e performance das implementações.

Um compilador é composto por duas fases, que são respectivamente implementadas pelos módulos de *frontend* e *backend*. O *frontend* possui quatro subfases: analisador léxico, analisador sintático, analisador semântico e gerador de código. Estas fases e subfases são detalhadas na Seção 1.4. Neste trabalho foram implementados os analisador léxico e analisador sintático. Deixando a construção do analisador semântico e gerador de código para uma segunda etapa, como trabalho futuro.

1.2.1 Objetivos e metas secundários

Outros objetivos a serem atingidos para considerarmos que o objetivo principal foi alcançado:

- a) Seguir as especificações da linguagem presente no Free Pascal: Reference Guide (CANNYET, 2012) e do Delphi Language Guide (EMBARCADERO, 2012).
- b) Construir o analisador léxico e sintático do frontend.
- c) As documentações já existentes para Object Pascal devem servir ao LLVM-Pascal com o mínimo de adaptações;
- d) Permitir que programadores medianos consigam entender, documentar, corrigir, otimizar e evoluir o *frontend* do compilador;
- e) Criar o LLVM-Pascal com muito menos linhas de código, o compilador de referência Free Pascal possui 255 kloc;

1.3 JUSTIFICATIVA

A linguagem escolhida como entrada do compilador e sua implementação foi o Object Pascal, para a qual há uma grande base de aplicações, bibliotecas, utilitários e componentes, muitos com fonte aberto (TORRY, 2012).

Além disso a estrutura da linguagem Pascal, com sintaxe mais simples e coerente que outras linguagens, foi criada para facilitar a implementação de compiladores levando a um projeto mais estruturado e simples do compilador. A linguagem Object Pascal será apresentada no Capítulo 2.

Em 2005 havia mais de 1,7 milhão de programadores Object Pascal. O dialeto Object Pascal que se tornou o padrão de fato de mercado foi o Object Pascal usado no Delphi. Apenas dois compiladores atualmente podem compilar tais aplicações, um comercial (Delphi, fonte fechado que até a versão XE2 é desenvolvido em assembly x86 e C (EMBARCADERO, 2012) e outro *open source* (Free Pascal, desenvolvido em Pascal).

Considerando o Free Pascal, por que então criar outro compilador Object Pascal *open source*? A primeira razão é que não existe um compilador para Object Pascal usando o LLVM.

Mas o grande motivador é o tamanho e a complexidade do compilador Free Pascal,

que ao longo de mais de 15 anos de desenvolvimento e contribuições de 55 desenvolvedores ativos, tem hoje mais de 255 kloc (OHLOH, 2012), o que o torna inviável como material de estudo e difícil de se entender mesmo para programadores Object Pascal experientes. A proposta do LLVM-Pascal é ter uma funcionalidade similar, com muito menos linhas de código. A ideia é que o LLVM-Pascal seja para o Free Pascal/Delphi, o que o Minix é para o Linux/Unix, e ainda que seja capaz de ser usado nos mesmos projetos onde o Free Pascal e o Delphi podem ser usados.

1.4 ESCOPO DO TRABALHO

O processo de compilação, bem estruturado, é composto por fases definidas e implementadas por módulos específicos do compilador. As duas principais fases são chamadas de Análise e Síntese, que são respectivamente implementadas pelos módulos de *frontend* e *backend*. Conforme mostra a Figura 3.

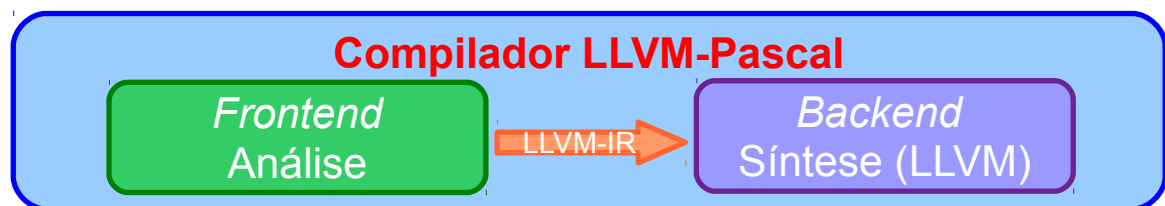


Figura 3 - As duas fases principais de um compilador

A fase de análise avalia a estrutura léxica e gramatical do programa fonte gerando uma representação em árvore desta estrutura, chamada AST (Abstract Syntax Tree) e uma representação tabular das estruturas de dados usadas pelo programa, chamada Tabela de Símbolos (ST – Symbol Table). A partir da AST e da ST é gerada uma representação intermediária (IR – Intermediate Representation), que nada mais é que a serialização da AST e da ST. Se o *frontend* detectar erros (léxicos, sintáticos ou semânticos) no programa fonte ele deverá emitir mensagens de erro para que o programador faça as correções necessárias.

A partir da IR, ou LLVM-IR no caso do LLVM, a parte de síntese gerará o código objeto, usando diversas subfases. A parte de síntese é inteiramente implementada pelo LLVM. As subfases do *frontend* e do *backend* são mostradas a Figura 4.

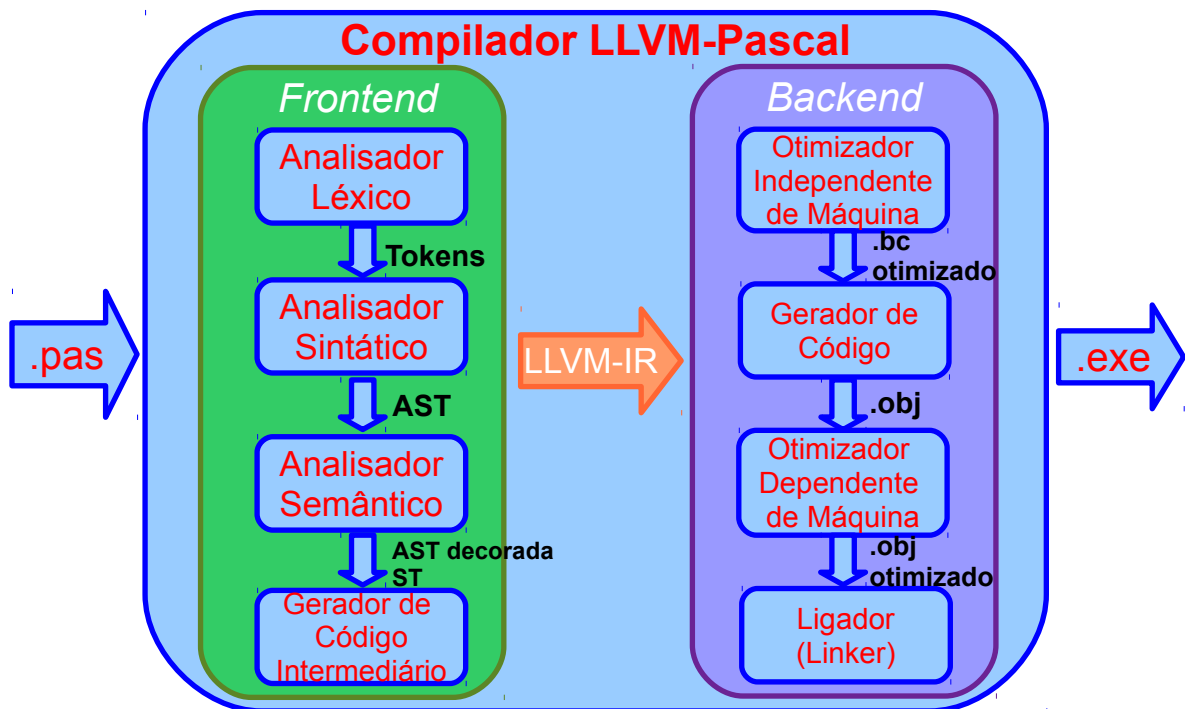


Figura 4 - Fases do compilador Object Pascal usando LLVM

- a) **Analizador Léxico (Lexer):** Lê o fluxo de caracteres que compõe um programa fonte e separa as palavras em lexemas (*tokens*);
- b) **Analizador Sintático (Parser):** Utiliza os *tokens* para produzir uma representação em árvore (AST), que mostra a estrutura gramatical da sequência de *tokens*;
- c) **Analizador Semântico (Analyser):** Verifica a consistência semântica do programa fonte com a definição da linguagem. Faz a verificação dos tipos das estruturas de dados usadas pelo programa e salva na AST e na ST, para uso subsequente durante a geração do código intermediário;
- d) **Gerador de Código Intermediário:** Gera um programa para uma máquina abstrata, com *código de três endereços*, que consiste em uma sequência de instruções tipo *assembly* com três operandos por instrução. No caso do LLVM o código intermediário chama-se *bitcode* (.bc) ou LLVM-IR;
- e) **Otimizador Independente de Máquina:** Faz várias transformações no código intermediário com o objetivo de produzir um código mais rápido e menor.
- f) **Gerador de Código Objeto:** Recebe o *bitcode* e o mapeia para um código

objeto de determinada máquina/processador.

- g) **Otimizador Dependente de Máquina:** Faz transformações adicionais no código objeto para tirar o máximo de proveito das características específicas do processador alvo.
- h) **Ligador (Linker):** Reúne todos os objetos necessários para a geração do programa executável final e resolve todas as questões de endereçamento ainda pendentes.

As fases “a”, “b”, “c” e “d”, que correspondem ao *frontend*, serão detalhadas respectivamente nos Capítulos 3, 4, 5, 6. Por serem alvo de codificação receberão, cada uma um capítulo próprio, descrevendo as estratégias e técnicas adotadas na implementação. Já as fases “e”, “f”, “g”, e “h”, que ocorrem no *backend*, serão explanadas de forma um pouco mais sucinta, no Capítulo 8, tendo em vista que já estão implementadas no LLVM.

1.5 AMBIENTE DE DESENVOLVIMENTO

O usuário do LLVM-Pascal usará o Lazarus, um poderoso IDE (Integrated Development Environment) licenciado como GPL (General Public License), para desenvolvimento de suas aplicações multiplataformas. Com a utilização de códigos *open source* disponibilizados pela Internet: *backend* LLVM, RTL e FCL do Free Pascal e LCL e IDE do Lazarus será possível construir um sofisticado sistema para desenvolvimento de aplicações multiplataforma, conforme mostra a Figura 5.

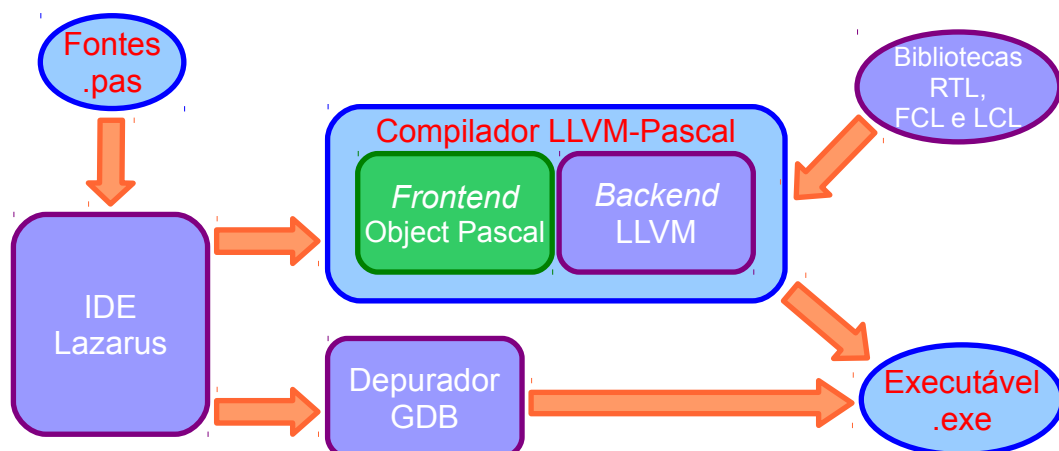


Figura 5 – Sistema de desenvolvimento com LLVM-Pascal

1.6 DIFERENCIAL DO PROJETO

Um diferencial do compilador Object Pascal usando LLVM é que o *frontend*, por ser implementado de forma modular e totalmente desacoplada do *backend*, poderá ter seu *output* (IR) redirecionado para outros *backends*. Facilitando a criação de outros compiladores Object Pascal a partir do mesmo *frontend*. Existem alguns *backends* open source ou *freeware* disponíveis que implementam a mesma filosofia de desacoplamento e código intermediário que o LLVM da Apple (LLVM, 2012):

- a) O Open64 da AMD;
- b) O EkoPath da PathScale e
- c) O Phoenix da Microsoft.

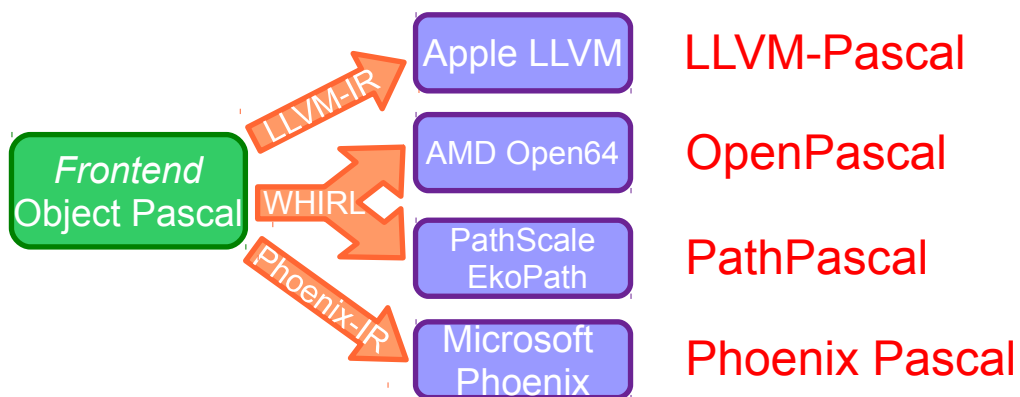


Figura 6 – O *frontend* Object Pascal sendo redirecionado para outros *backends* open source ou *freeware*

Tais compiladores usam outras representações intermediárias, conforme mostra a Figura 6, e geram códigos objeto muito diferentes do LLVM-Pascal original, levando para patamares com performances diferentes.

1.7 DESENVOLVIMENTO COLABORATIVO

O desenvolvimento do LLVM-Pascal está *on-line* através do GoogleCode (LLVM-PASCAL, 2012). Com isso ficam facilitados e assegurados a divulgação e o desenvolvimento do projeto num site central. Outros alunos ou colaboradores podem usar e participar do

projeto bem como propor melhorias. Ver Figura 7.



Figura 7 – Página inicial do site do LLVM-Pascal no GoogleCode

O GoogleCode (GOOGLECODE, 2012) é similar ao SourceForge (SOURCEFORGE, 2012), porém muito mais fácil de usar. Possui recursos de desenvolvimento colaborativo *on-line*, via Internet, para sistemas *open source* com:

- a) Subversion: controle de versão dos programas-fonte e documentação;
- b) *Download* de arquivos do projeto: fontes, executáveis e documentação;
- c) Páginas *wiki* para documentação *on-line*;
- d) *Bugtracking* e controle de demandas e mudanças;
- e) *Links*;
- f) Fórum de discussões;
- g) Estatísticas de acesso ao site.

1.8 METODOLOGIA DE DESENVOLVIMENTO

Como o *frontend* foi desenvolvido em Object Pascal e será *self-hosted*, o Free Pascal foi usado como compilador de *bootstrap*. Isso quer dizer, que a primeira versão do compilador será gerada pelo Free Pascal, que é o processo de *bootstrapping*. Mas as demais versões serão compiladas pelo próprio LLVM-Pascal, que é o processo de *self-hosting*. Os processos de *bootstrapping* e *self-hosting* são formalmente representados usando *T-diagrams* (TERRY, 1997).

Um *T-diagram* representa um tradutor a partir de um bloco no formato de um “T”. No braço esquerdo é indicada a linguagem fonte usada como entrada do tradutor. No braço direito é indicada a linguagem objeto gerada como saída do tradutor. No pé do “T” é informada a linguagem em que o tradutor está implementado. Ver Figura 8.



Figura 8 – Bloco de um *T-diagram*

No processo de *bootstrapping* do LLVM-Pascal, o *T-diagram* é composto por 3 tradutores, como mostra a Figura 9. O primeiro é o programa-fonte do LLVM-Pascal que traduz Object Pascal (arquivos .pas) em LLVM-IR ou *bitcode* (arquivos .bc) e é escrito em Object Pascal. O segundo tradutor é o compilador de *bootstrapping*, no caso o Free Pascal, que traduz Object Pascal para código executável, sendo ele mesmo um programa executável. O terceiro é o tradutor que é gerado: um executável que traduz Object Pascal em *bitcode*.

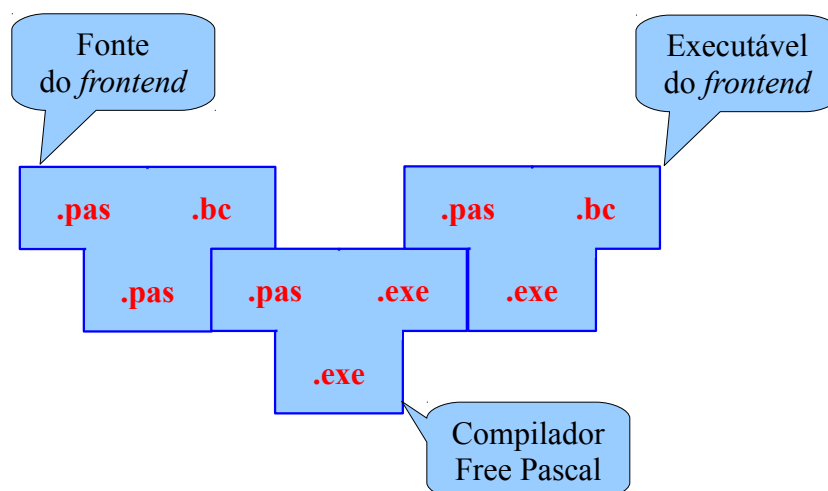


Figura 9 – *Bootstrapping* do LLVM-Pascal, usando Free Pascal

O próprio LLVM é *bootstrapped* pelo GCC conforme mostra a Figura 10.

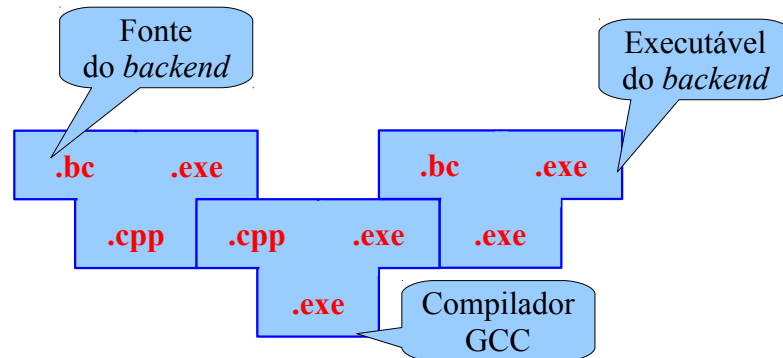


Figura 10 – *Bootstrapping* do backend LLVM, usando GCC

Após o *bootstrapping* do *frontend* e do *backend* é obtido o compilador LLVM-Pascal como um executável, conforme mostra a Figura 11.

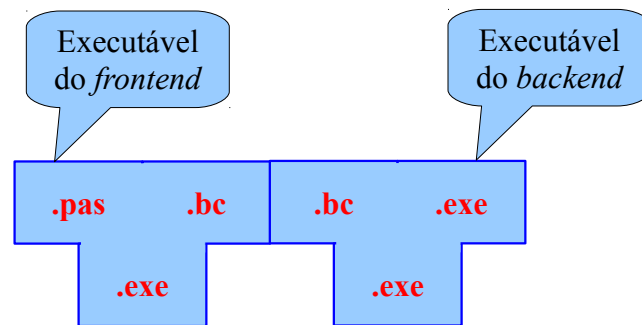


Figura 11 - O novo compilador LLVM-Pascal após o *bootstrapping*

Após a montagem do compilador ele deverá ser capaz de *self-hosting*, ou seja, compilar a si mesmo o que será feito sempre que avançar para versões mais novas, conforme mostra a Figura 12.

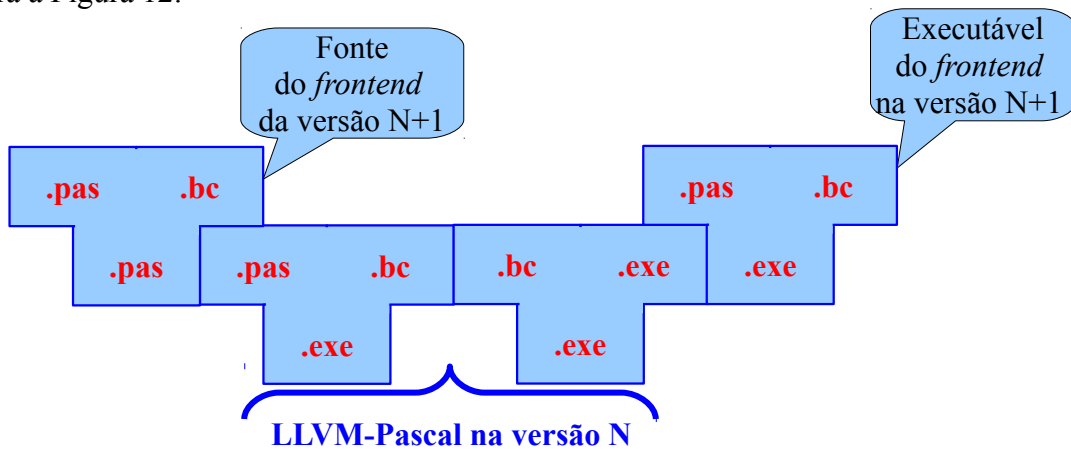


Figura 12 – *Self-hosting* do compilador migrando de uma versão N para N+1

2 OBJECT PASCAL

Este capítulo serve como referência para a linguagem Object Pascal que é implementada para o compilador LLVM-Pascal. Aborda as construções do Pascal e lista os tipos de dados suportados. Não se trata, no entanto, de um tutorial da linguagem Pascal.

Também é abordado o IDE (ambiente de desenvolvimento integrado) Lazarus *open source*, que suporta o desenvolvimento na linguagem Object Pascal. Há uma relação muito estreita entre o Object Pascal e o IDE.

Pascal foi definido em 1970 pelo Prof. Niklaus Wirth do Instituto de Tecnologia da Suíça (Zurique) (WIRTH, 1971), como extensão do ALGOL, linguagem de programação voltada para computação científica. Em um congresso em Zurique, um comitê internacional designou o ALGOL como uma linguagem independente de plataforma. Isto deu mais liberdade para as características que eles poderiam colocar na linguagem, mas também tornou mais difícil a escrita de compiladores para ela. A carência de compiladores em muitas plataformas, combinada com a falta de muitos tipos de dados básicos tais como caracteres, fez com que o ALGOL não fosse amplamente aceito e acabasse quase que totalmente abandonado. Restando apenas a implementação em computadores de grande porte da Unisys.

A especificação do Pascal original foi publicada em 1971. O nome da linguagem foi dado em homenagem ao filósofo e matemático francês Blaise Pascal, que inventou uma máquina de calcular mecânica. Pascal é bastante orientado a tipos de dados, dando ao programador a capacidade de definir tipos de dados personalizados. Com esta liberdade veio a rígida checagem de tipos, que garantiu que os dados não seriam usados de forma incorreta. Pascal pretendia ser uma linguagem educacional e foi amplamente adotada como tal. É uma linguagem de escrita mais livre. Diferente do FORTRAN, que exige que certos elementos da linguagem comecem em determinadas colunas do texto. Assim os estudantes não tinham que se preocupar com formatação. Além disso, Pascal se parece muito com a linguagem ALGOL da qual herdou as características principais, tornando muito fácil o entendimento do código escrito com ela.

Object Pascal é a extensão orientada a objetos da linguagem Pascal que foi desenvolvida pela Apple em parceria com Wirth, inventor do Pascal. A Apple usou o Object Pascal para desenvolver o primeiro computador comercial com interface gráfica o

revolucionário Lisa (nome da primeira filha de Steve Jobs), cujos conceitos foram copiados da Xerox. Tanto o sistema operacional, como a interface gráfica e as aplicações de escritório do Lisa (planilha eletrônica e editor de textos) foram escritos em Object Pascal. Posteriormente a Apple abandonou o Object Pascal em favor do “C” ao lançar o Macintosh em 1984. Inicialmente existiam poucas modificações na sintaxe de Object Pascal em relação ao Pascal. Um novo tipo de dados foi adicionado: *object*. No qual era possível especificar uma lista de procedimentos e funções, referenciados como métodos, para cada tipo de objeto. Estes métodos definiam as ações que objetos daquele tipo podem realizar. Bem como a lista de atributos inerentes a cada objeto. Também definia uma hierarquia entre esses objetos através do conceito de herança (CRAIG, 1993).

2.1 BORLAND PASCAL

Em novembro de 1983 a Borland lançou o Turbo Pascal, iniciando suas atividades como fornecedora de ferramentas e ambientes de desenvolvimento. O Turbo Pascal era um compilador comercial, desenvolvido por Anders Hejlsberg, muito mais simples, mais rápido e mais barato (US\$ 49,99) que qualquer um existente na época. Anos mais tarde Hejlsberg foi o arquiteto chefe e criador do .NET da Microsoft, que importou muitas das ideias e conceitos dos compiladores Object Pascal da Borland. O Turbo Pascal introduziu um IDE, onde era possível fazer um ciclo básico de desenvolvimento em segundos: editar o código, executar o compilador, observar os erros e voltar para as linhas que continham esses erros, apertando umas poucas teclas do computador. Este procedimento era inédito na época. O compilador Turbo Pascal foi um sucesso de vendas, sendo o compilador mais vendido de todos tempos. O que tornou o Pascal bastante popular em PC's na década de 80.

2.2 CARACTERÍSTICAS

O Object Pascal herdou a estrutura sintática do Pascal estruturado. Assim como o C++ herdou a estrutura sintática do ANSI-C. Em ambos os casos houve uma mudança radical com o acréscimo do paradigma orientado a objetos. Com esse paradigma muda radicalmente o estilo de programação.

As seguintes características foram acrescentadas à linguagem Pascal para obter a

linguagem Object Pascal:

- a) Definição de novas classes por meio da palavra reservada ***class*** ou ***object***;
- b) Encapsulamento: Obtido usando os modificadores de acesso: ***public***, ***protected*** e ***private***. Outro recurso poderoso que a linguagem apresenta para a obtenção de encapsulamento são as ***properties***, que definem métodos acessores de forma elegante e implícita;
- c) Herança;
- d) Polimorfismo (*Late Binding*): O método a ser chamado é conhecido apenas em tempo de execução e não em tempo de compilação (*Early Binding*), dependendo do tipo do objeto para o qual o método foi chamado. Obtemos polimorfismo com o uso das palavras reservadas ***virtual*** e ***override***;
- e) Métodos abstratos e interfaces: Uma característica interessante do Object Pascal é que uma classe pode ser definida com métodos abstratos ou sem corpo, que só são efetivamente implementados em classes descendentes;
- f) Exceções: O mecanismo de exceções de Object Pascal é muito semelhante ao do Java e baseia-se em quatro palavras reservadas: ***try***, ***except***, ***finally*** e ***raise***;
- g) Suporte a *Multithreading*.

2.3 IMPLEMENTAÇÃO

Aplicações em Object Pascal além do paradigma OO (orientação a objetos) usam o paradigma *Rapid Application Development* (RAD). Para a implementação desse paradigma há uma forte relação entre a linguagem e o IDE. Todos os exemplos de programação usados neste trabalho usam o Lazarus como IDE padrão para Object Pascal. Assim uma aplicação Object Pascal RAD é composta por: um programa principal ou arquivo de projeto (.lpr, Lazarus Project), formulários (.lfm, Lazarus Form) que descrevem o *layout* das telas gráficas, num formato texto similar à inicialização de classes em Pascal e *units* (.pas, fonte Pascal) com a programação dos eventos e a lógica do negócio. A organização e edição desses arquivos é feita de forma semi-automatizada pelo IDE, padronizando o desenvolvimento e concedendo grande produtividade aos programadores.

2.3.1 Estrutura Geral de uma Unit

Uma *unit* se divide em duas seções:

- a) Interface (***interface***): Nela são declaradas tipos, classes, variáveis, constantes e métodos. Essa é a parte visível da *unit* para outras *units* que a usem. Para usar uma *unit*, outra *unit* faz referência à primeira usando a palavra reservada ***uses***.
- b) Implementação (***implementation***): Essa seção só é acessada de dentro da própria *unit*. Nela estão as implementações dos métodos declarados na interface e, eventualmente, outros de uso local apenas.

Em Object Pascal mais de uma classe pode ser definida na mesma *unit*. Isso permite implementar o conceito de classes amigas existente em C++. Ou seja, as classes dentro da mesma *unit* poderão acessar livremente as implementações umas das outras, desde que não sejam usados modificadores de acesso ***strict***.

2.4 RTL, FCL E LCL

Os recursos usados pelos programadores Object Pascal são oriundos das bibliotecas, que nada mais são que conjuntos de *units*. Essas *units* foram disponibilizadas pelas equipes do Free Pascal e Lazarus com licença LGPL (Lesser General Public License) e possuem centenas de classes com milhares de métodos. Para fins de organização elas foram dispostas em 3 níveis de abstração RTL, FCL e LCL, conforme mostra a Figura 13.

- a) RTL (Run Time Library ou Biblioteca de Tempo de Execução): Possui o conjunto de *units* de nível mais baixo. Normalmente funções matemáticas e tratamentos de *strings*. Nesse nível estão os códigos de programação mais associados ao paradigma estruturado.
- b) FCL (Free Component Library): É o conjunto de *units* de definem classes não visuais mais abstratas que a RTL, tais como: listas, filas, *arrays* associativos, *threads*, *hashs* e acesso básico aos bancos de dados relacionais. Nesse nível residem os códigos de programação mais associados ao paradigma OO.
- c) LCL (Lazarus Component Library): Abriga os componentes visuais (GUI) para qualquer plataforma (Windows, Linux, Mac OS, WinCE, Android e iOS), tais

como: janelas, botões, combos, menus, relatórios, etc. Bem como a integração destes com bancos de dados relacionais. Nesse nível temos o paradigma visual orientado a eventos, também chamado RAD.

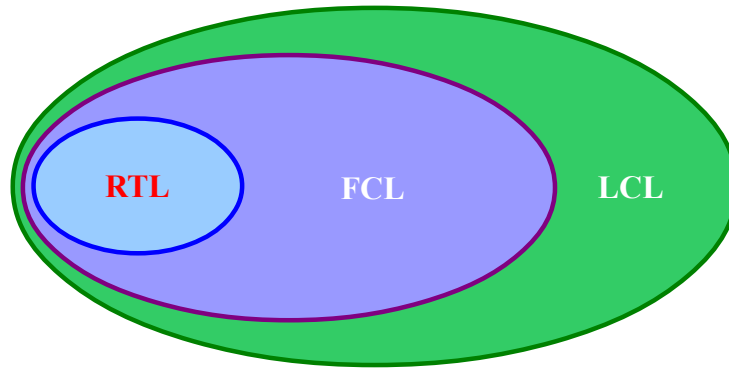


Figura 13 – As bibliotecas RTL, FCL e LCL e seus níveis de abstração

2.5 UM EXEMPLO DE PROGRAMA EM OBJECT PASCAL

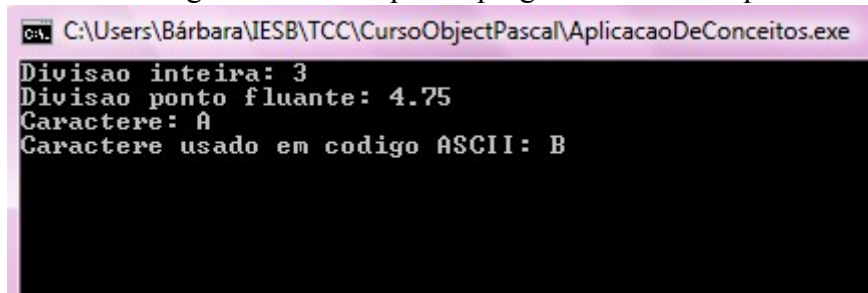
O exemplo, conforme as Figuras 14 e 15, mostra um programa em Pascal, utilizando alguns tipos primitivos da linguagem e sua saída no terminal.

```

Editor de Código
AplicacaoDeConceitos.lpr
1  program AplicacaoDeConceitos;
.
.
.  const
.    f = 2;
5  var
.    a: integer;
.    b: integer;
.    c: double;
.    d: char;
10   e: char;
.
.  begin
.    a := 7;
.    b := 2;
15   c := 9.5;
.    d := 'A';
.    e := #66;
.    WriteLn('Divisao inteira: ', a div f);
19   WriteLn('Divisao ponto fluante: ', c/b :4:2);
20   WriteLn('Caractere: ', d);
.    WriteLn('Caractere usado em codigo ASCII: ', e);
.    ReadLn;
.  end.
24

```

Figura 14 – Exemplo de programa Pascal simples



```
CA: C:\Users\Bárbara\IESB\TCC\CursoObjectPascal\AplicacaoDeConceitos.exe
Divisao inteira: 3
Divisao ponto fluante: 4.75
Caractere: A
Caractere usado em codigo ASCII: B
```

Figura 15 – Saída do exemplo do programa simples

2.6 CARACTERÍSTICAS GERAIS

Embora tenha evoluído ao longo dos anos com contribuições de diversas instituições: Apple, Borland e Universidade de San Diego entre outras, o Pascal manteve algumas características:

- a) Sintaxe fácil de ler e entender. Ideal para manutenção de sistemas e para o ensino;
- b) Compilação em um passo, mais rápida e simples que a compilação para outras linguagens;
- c) Tipagem forte, permitindo ao compilador achar erros semânticos mais facilmente;
- d) Paradigma estruturado e imperativo;
- e) *Case insensitive*, ou seja, maiúsculas e minúsculas não fazem diferença;
- f) Procedimentos aninhados;
- g) Funções de entrada e saída *built-in* e fáceis de ser usadas.

2.7 ELEMENTOS DA LINGUAGEM

2.7.1 Identificadores

Identificadores são nomes definidos pelo programador para especificar constantes, variáveis, procedimentos, funções, tipos, *units* e programas.

Os identificadores podem ter de 1 a 127 caracteres significativos. O primeiro caractere de um identificador deve ser obrigatoriamente uma letra ou *underscore* “_”. Os demais podem ser letras, números ou *underscore*. Alguns exemplos de identificadores válidos

são:

```
a123
_abcd
ABC_123
```

Pascal é *case insensitive*, ou seja, os dois identificadores a seguir representam o mesmo dado:

```
MinhaFuncao e
minhafuncao
```

2.7.2 Palavras Reservadas

Palavras reservadas são parte de uma linguagem de programação e não podem ser redefinidas pelo programador como um identificador. Entretanto usando-se um “E comercial” “&” como prefixo de uma palavra reservada é possível usá-la como um identificador. Dessa forma é possível declarar, por exemplo, `&function` como identificador válido.

2.7.3 Comentários

Um programa pode ter comentários, que são partes do código ignoradas pelo compilador. Os comentários servem para explicar peças do código. Há três maneiras para especificar comentários:

```
(* Comentários em múltiplas linhas usados nos primeiros
   compiladores Pascal *)
{ Comentários em múltiplas linhas introduzidos pelo Turbo
  Pascal }
// Comentários em uma linha do Object Pascal
```

Um abre chave “{” seguido de um cifrão “\$” torna-se uma diretiva do compilador. Diretivas do compilador são instruções que devem ser interpretadas durante a compilação do programa. Por exemplo, a diretiva `{IFDEF LINUX}` indica que o código a seguir só será compilado se a compilação estiver ocorrendo em um sistema operacional Linux.

2.7.4 Variáveis

Variáveis são locais de memória identificados por um nome (identificador), que podem armazenar um tipo de dado. O local onde a variável é armazenada depende de onde ela é declarada.

a) Variáveis globais são declaradas em uma *unit* ou programa, fora de um método. São armazenadas em locais fixos na memória e disponíveis durante todo o tempo de execução do programa.

b) Variáveis locais são declaradas dentro de um método. São armazenadas na pilha do programa e estão disponíveis apenas no escopo onde são declaradas.

As variáveis, tanto globais quanto locais, são declaradas em uma seção denominada *var*.

A declaração consiste de um identificador, seguido de dois pontos “:” e do tipo da variável. Por exemplo:

```
var
  a : integer;
  c : char;
  s : string;
```

Em linguagens de programação, *strings* são cadeias de caracteres. Por exemplo 'João da Silva', 'avião' e '12345' são *strings*. As cadeias de caracteres são delimitadas por apóstrofes.

2.7.5 Constantes

Constantes são dados que não se alteram durante a execução do programa. Dados constantes podem ser referenciados através de identificadores, desde que tenham sido declarados antes em uma seção do programa chamada *const*. Podemos declarar dois tipos de constantes: as constantes comuns e as constantes tipadas.

As constantes comuns são declaradas usando um identificador seguido de um sinal igual “=” e de uma expressão que resulte em um valor constante. Essas expressões são avaliadas em tempo de compilação. As seguintes declarações de constantes são válidas:

```
const
    a = 2;
    c = '4';
    s = 'constante string';
```

Constantes tipadas especificam o tipo da constante quando são declaradas. Por exemplo:

```
const
    a : integer = 2;
    c : char = 'A';
```

As constantes tipadas, quando declaradas dentro de métodos, funcionam como variáveis estáticas.

2.7.6 Tipos Ordinais

Estes tipos armazenam valores discretos, baseiam-se no conceito de ordem ou sequência. É possível comparar dois valores, determinar qual o valor seguinte ou anterior a um dado valor e ainda calcular o maior ou menor valor possível. Na Tabela 1 são apresentados os tipos ordinais em uma tabela com seus nomes, faixa de valores e tamanhos:

Tabela 1 – Tipos Ordinais

Tipo	Faixa de valores	Tamanho em bytes
Byte	0..255	1
Shortint	-128..127	1
Word	0..65535	2
Smallint	-32768..32767	2
Longint	-2147483648..2147483647	4
Integer	-2147483648..2147483647	4
Cardinal	0..4294967295	4
LongWord	0..4294967295	4
Int64	-9223372036854775808.. 9223372036854775807	8
QWord ou UInt64	0..18446744073709551615	8

2.7.7 Tipos Lógicos

O Object Pascal suporta tipos lógicos (*Boolean*) com os valores pré-definidos *True* e *False*. Além do *Boolean*, existem o *ByteBool*, *WordBool* e *LongBool*. Estes três últimos tipos raramente são usados. As expressões lógicas são, por padrão, avaliadas como *short-circuit* de forma que, quando o resultado é conhecido o resto da expressão não é mais avaliada. Por exemplo: se *B* é *False* e *Funcao* retorna um *boolean*, na expressão: *B and Funcao*, *Funcao* nunca é executada, uma vez que o resultado da expressão já é conhecido (*False*).

2.7.8 Tipos Enumerados

Tipos enumerados ou enumerações são um tipo ordinal definido pelo usuário. Uma enumeração é uma lista de valores e é declarada em uma seção *type*. Por exemplo:

```
type
    DiasSemana = (Domingo, Segunda, Terca, Quarta, Quinta,
                  Sexta, Sabado);
    Cores = (Vermelho, Amarelo, Laranja, Verde, Cianeta,
            Azul, Violeta);
```

2.7.9 Tipos Subrange

O tipo *subrange* corresponde a uma faixa de valores de um tipo ordinal. Para definir um tipo *subrange*, devem se especificados seus limites: o valor mínimo e máximo do tipo, separados por ponto ponto “..”. Por exemplo:

```
type
    Idade = 0..100;
```

Uma variável declarada com esse tipo só aceitará valores inteiros de 0 a 100.

2.7.10 Tipos Reais

Tipos reais representam valores numéricos contínuos de ponto flutuante com vários graus de precisão. O menos preciso é o *Single*, armazenado em 4 *bytes*. O *Double* é o mais

usado e é implementado em 8 *bytes*. Além desses há outros tipos com precisões diferentes que são suportados pelo coprocessador aritmético da CPU. Os tipos reais suportados são relacionados, conforme mostra a Tabela 2.

Tabela 2 – Tipos Reais

Tipo	Faixa de valores	Tamanho em bytes
Single	1.5e-45 .. 3.4e+38	4
Double	5.0e-324 .. 1.7e+308	8
Real	5.0e-324 .. 1.7e+308	8
Extended	3.4e-4932 .. 1.1e+4932	10
Comp	$-2^{63}+1$.. $2^{63}-1$	8
Currency	-922337203685477.5808.. 922337203685477.5807	8

2.7.11 Tipos Character

O tipo *char* tem tamanho 1 *byte* e contém um caractere ASCII. Um *char* pode ser especificado por um caractere entre apóstrofes: 'a', 'C' ou '8'. Ou ainda informando seu valor na tabela ASCII, precedido do símbolo jogo-da-velha “#”. Por exemplo:

```
var
  a : char = 'B';
  c : char = #65; // 65 é o código ASCII de 'A'
```

Um caractere Unicode também pode ser representado, analogamente, usando o tipo *WideChar*.

2.7.12 Tipos String

Um tipo *string* é capaz de armazenar textos, ou seja, cadeias de caracteres. Uma *string* é especificado por um texto delimitado por apóstrofes. Por exemplo:

```

var
  s : string = 'Um texto';
  t : string = 'Uma string com apostrofes '' dentro';

```

Para compatibilização com versões mais antigas do Pascal, permitir a internacionalização das aplicações e a interoperabilidade com outras linguagens, em especial o “C”, o Object Pascal suporta uma grande variedade de tipos *string*, conforme mostra a Tabela 3.

Tabela 3 – Tipos String

Tipo	Alocação de Memória	Tamanho em bytes	Tabela
String	dinâmica e automática	Até 2 GB	ASCII
String[N]	estática	N, N <=255	ASCII
ShortString	estática	255	ASCII
AnsiString	dinâmica e automática	Até 2 GB	ASCII
WideString	dinâmica e automática	Até 1 GB	Unicode
PChar	dinâmica e manual	Até 2GB	ASCII
PWideChar	dinâmica e manual	Até 1 GB	Unicode

2.8 COMANDOS

Apresentamos sucintamente os principais comandos da linguagem que servem para escrever algoritmos. Os comandos se baseiam em palavras chave e símbolos que formam uma sequência finita e ordenada de operações que o programa deve executar para alcançar seus objetivos.

O Object Pascal faz parte da família de linguagens de programação que descende do ALGOL. Modula, Oberon e Ada são outros exemplos de linguagens pertencentes a esta família. Assim sendo, a maior parte das estruturas sintáticas apresentadas a seguir é muito semelhante nas linguagens pertencentes a esta família.

2.8.1 Atribuição

Uma atribuição é especificada pelo operador *dois pontos igual* “:=”. Uma notação diferente, para aqueles que estão habituados a usar linguagens da família “C”. O operador

igual “=” é usado para fazer testes de comparação. Por exemplo:

```
a := 2;
s := 'isto é uma string';
```

Nas duas linhas acima foram atribuídos valores às variáveis.

2.8.2 Operadores e Expressões

Os diversos tipos de operadores são apresentados conforme mostram as Tabelas 4, 5 e 6 respectivamente.

Tabela 4 - Operadores aritméticos

Operador	Operação
+	Adição
-	Subtração
/	Divisão
*	Multiplicação
**	Exponenciação
div	Divisão de inteiros
mod	Resto da divisão inteira

Tabela 5 - Operadores lógicos

Operador	Operação
not	Negação
and	E
or	Ou
xor	Ou exclusivo
shr	Desloca bits para direita
shl	Desloca bits para a esquerda

Tabela 6 - Operadores relacionais

Operador	Operação
=	Igual
<>	Diferente
<	Menor que
>	Maior que
<=	Menor ou igual
>=	Maior ou igual
in	Pertence

Expressões envolvendo operadores relacionais retornam True ou False. Exemplos de expressões utilizando operadores *boolean* e relacionais:

```
(a > 3) and (a < 10)
not(a <= 10)
```

2.8.3 Chamadas de métodos e procedimentos

Trechos de código podem ser fatorados em procedimentos ou métodos e chamados posteriormente como se fossem comandos da linguagem. Este recurso existe na maioria das linguagens imperativas. Em Object Pascal há três maneiras diferentes de fazer chamadas a procedimentos:

a) Chamada a procedimento normal;

```
ClrScr; // procedure para limpar tela do console
WriteLn; // procedure para pular uma linha no console
```

b) Chamada a um método de um objeto;

c) Chamada a uma variável procedural.

2.9 ESTRUTURA BÁSICA DE UM PROGRAMA

Um programa em Object Pascal é dividido em seções, descritas abaixo:

a) **Program**: Identificação do programa. Inicia pela palavra-chave *program* seguida de um identificador e finalizando com ponto e vírgula “;”;


```
program exemplo;
```

- b) **Uses:** Declaração das unidades utilizadas (*units*). As *units* (termo usado em Pascal para bibliotecas), são formadas por um conjunto de declarações, procedimentos e funções que podem ser usadas por um programa ou outras *units*. Por exemplo, a *unit* Crt contém o comando ClrScr que limpa o console. Para que um programa possa utilizar um procedimento pertencente a uma determinada *unit* é necessário que tal *unit* seja relacionada na seção **uses**;

```
uses Crt;
```

- c) **Type:** Definições de tipos de dados. O Object Pascal tem tipos pré-definidos, tais como os apresentados na seção 2.8. O programador também pode definir novos tipos nesta seção. Por exemplo, para criar o tipo DiasSemana, temos a sintaxe:

```
type
```

```
    DiasSemana = (Domingo, Segunda, Terca,  
                  Quarta, Quinta, Sexta, Sabado);
```

- d) **Const:** Declaração de constantes. Nesta seção são declarados as constantes usadas pelo programa ou *unit*;
- e) **Var:** Declaração de variáveis. Aqui são declaradas as variáveis globais do programa ou *unit*;
- f) **Procedure/Function:** Declaração de métodos, procedimentos e funções. São conjuntos de declarações e comandos com estrutura similar a um programa, são chamados também subrotinas ou subprogramas. Para que uma subrotina seja utilizada no programa ou *unit* ela deve ser declarada nesta seção.
- g) Bloco do programa principal. Esta seção contém os comandos que serão executados assim que o programa é invocado. O bloco principal inicia com a palavra **begin** e termina com a palavra **end.** seguida de um ponto. Ver Figura 16.

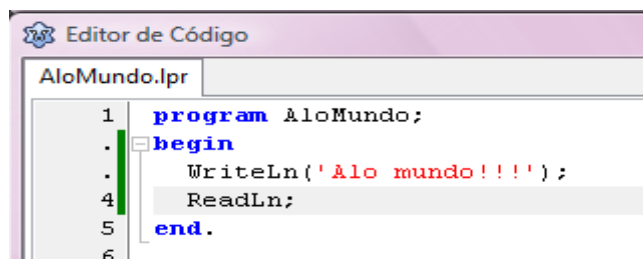


Figura 16 - Programa Alô Mundo

Para compilar e executar o programa tecla CTRL + F9. Se a compilação não tiver erros, uma janela conforme mostra a Figura 17 aparecerá. Tecle ENTER para retornar ao IDE.

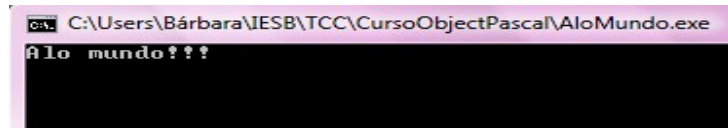


Figura 17 - Programa Alô Mundo executando

No bloco principal podemos notar o comando `Writeln('Alo Mundo');`. Este comando é usado para escrever algo na tela. Neste caso a sequência de caracteres 'Alo Mundo!!!' é mostrada no terminal.

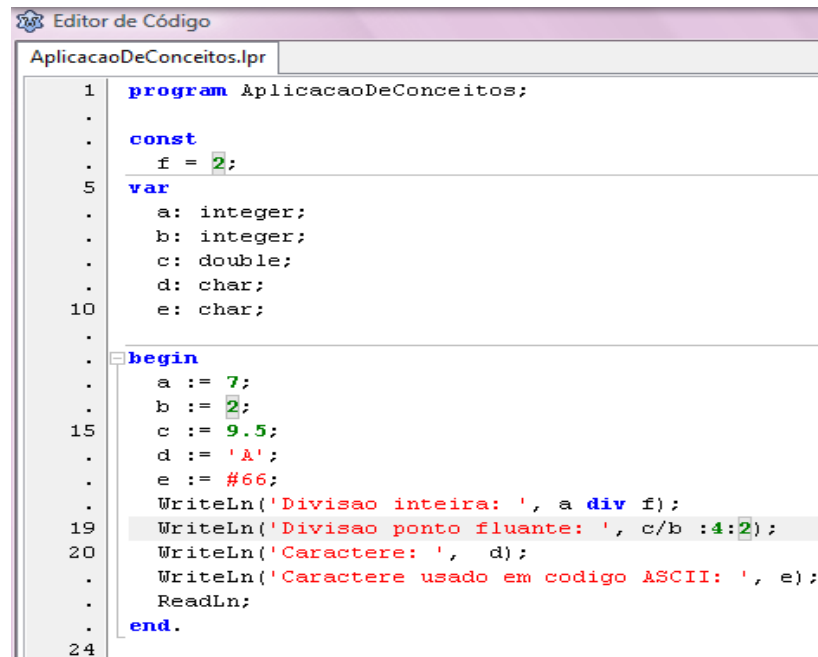
Neste programa o comando `Readln` aguarda o pressionamento da tecla ENTER, permitindo que possamos visualizar o resultado do programa no terminal.

2.10 ESCRIVENDO DADOS NO TERMINAL

A exibição de dados na tela é feita através dos comandos de saída. Há dois comandos de saída para o terminal:

```
Write(param1, param2 .. paramN);  
e  
Writeln(param1, param2 .. paramN);
```

Ambos enviam a lista de parâmetros para a tela, sendo que `Writeln` envia também caracteres que fazem o cursor se posicionar na próxima linha. A lista de parâmetros pode conter identificadores de variáveis, constantes ou expressões, que são avaliadas e o seu resultado é exibido em cadeias de caracteres. Cada elemento na lista de parâmetros é separado por vírgula. Ver Figura 18.



```

1  program AplicacaoDeConceitos;
.
.  const
.    f = 2;
5  var
.    a: integer;
.    b: integer;
.    c: double;
.    d: char;
10   e: char;
.
.  begin
.    a := 7;
.    b := 2;
15   c := 9.5;
.    d := 'A';
.    e := #66;
.    WriteLn('Divisao inteira: ', a div f);
19   WriteLn('Divisao ponto fluante: ', c/b :4:2);
20   WriteLn('Caractere: ', d);
.    WriteLn('Caractere usado em codigo ASCII: ', e);
.    ReadLn;
.  end.
24

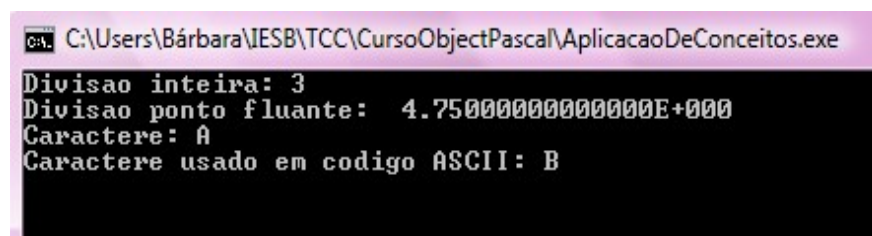
```

Figura 18 – Exemplo com Writeln

No código podemos observar a declaração de uma constante e de algumas variáveis. No bloco de principal, primeiramente as variáveis são inicializadas usando o operador de atribuição “:=”.

Sendo Object Pascal uma linguagem fortemente tipada, é necessário que os valores atribuídos às variáveis sejam compatíveis aos tipos informados na declaração de cada variável. Ou seja, deve-se atribuir um número inteiro a uma variável *integer*, um caractere a uma variável *char* e, um número real a uma variável *double*.

Em seguida é usado o comando `Writeln` para mostrar dados na tela. Ao executar esse programa, será escrito o seguinte resultado na tela, conforme mostra a Figura 19.



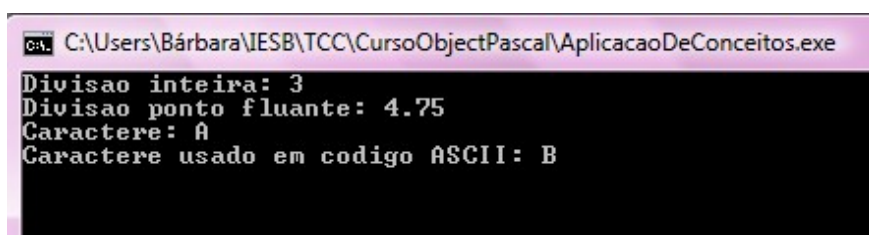
```

C:\Users\Bárbara\IESB\TCC\CursoObjectPascal\AplicacaoDeConceitos.exe
Divisao inteira: 3
Divisao ponto fluante: 4.750000000000000E+000
Caractere: A
Caractere usado em codigo ASCII: B

```

Figura 19 - Resultado do programa da Figura 18

Note que o resultado da divisão de ponto flutuante foi apresentado no formato que chamamos notação científica: 4.750000000000000E+000. O valor após o “E” correspondente ao expoente na base 10. Portanto, esse resultado equivale a 4.75×10^0 . Esse formato não é muito amigável, por isso o comando `Writeln` oferece uma maneira de formatar o resultado: `expressao : numchars [:decimais]`. `Numchars` representa a quantidade de caracteres e `decimais` a quantidade de dígitos após o símbolo decimal. Dessa forma, esta linha poderia ser modificada para: `writeln('Divisão ponto flutuante: ', c / b : 7 : 2);` E assim o resultado mostrado deve ser: divisão ponto flutuante: 4.75. Conforme mostra a Figura 20.



```

C:\Users\Bárbara\IESB\TCC\CursoObjectPascal\AplicacaoDeConceitos.exe
Divisao inteira: 3
Divisao ponto fluante: 4.75
Caractere: A
Caractere usado em codigo ASCII: B

```

Figura 20 – Saída formatada corretamente

2.11 ENTRADA DE DADOS

De uma maneira geral os programas tratam dados que são fornecidos pelos usuários. Esses dados devem ser armazenados em variáveis e são chamados de entrada. A entrada de dados pelo terminal é feita utilizando os comandos:

```

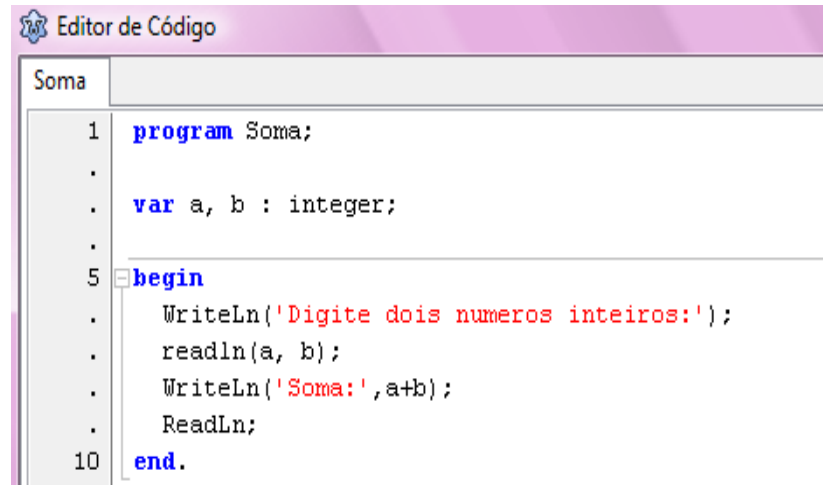
Read(param1, param2 .. paramN);
ou
Readln(param1, param2 .. paramN);

```

Da mesma forma que o comando `Writeln`, o comando `Readln` lê os dados da entrada padrão (teclado), armazena-os nas variáveis relacionadas nos parâmetros e salta para a próxima linha do terminal.

Os identificadores listados nos parâmetros devem ser separados por vírgulas. Durante a execução do programa, quando o fluxo encontra um comando `Read`, ele fica aguardando que o usuário digite uma certa quantidade de valores, correspondente à quantidade de

identificadores especificados no comando. Após cada valor, o usuário pode digitar um espaço ou teclar ENTER. A seguir um exemplo de entrada de dados com `ReadLn`. Ver Figura 21.



```

1  program Soma;
.
.  var a, b : integer;
.
5  begin
.    WriteLn('Digite dois numeros inteiros:');
.    readln(a, b);
.    WriteLn('Soma:',a+b);
.    ReadLn;
10 end.

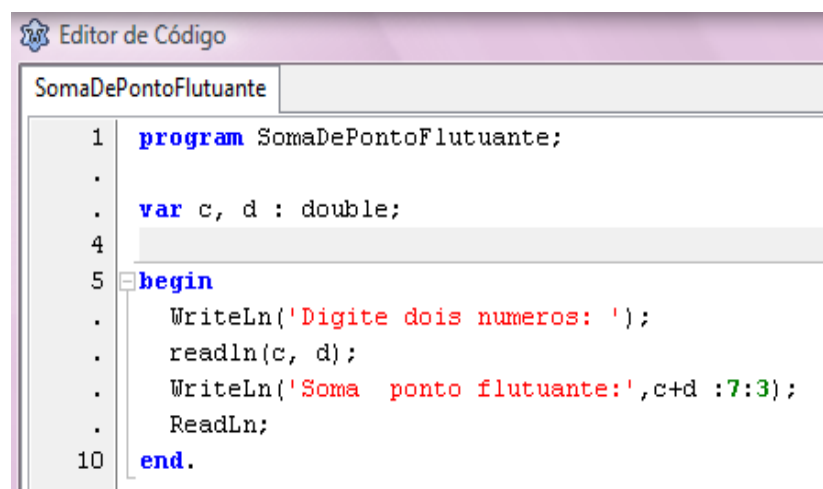
```

Figura 21 - Utilização de `ReadLn`

Foram declaradas duas variáveis inteiras: `a` e `b`. Mais de uma variável de um mesmo tipo pode ser declarada em uma única linha.

No bloco principal essas duas variáveis são lidas usando `ReadLn` e depois a sua soma é enviada para o terminal usando `WriteLn`.

O programa mostrado na Figura 22 efetua soma de números de ponto flutuante. Onde a saída foi adequadamente formatada.



```

1  program SomaDePontoFlutuante;
.
.  var c, d : double;
4
5  begin
.    WriteLn('Digite dois numeros: ');
.    readln(c, d);
.    WriteLn('Soma ponto flutuante:',c+d :7:3);
.    ReadLn;
10 end.

```

Figura 22 - Soma de ponto flutuante.

3 ANALISADOR LÉXICO

É a primeira fase do compilador, também chamada de *Scanner* ou *Lexer*, que faz a leitura do programa-fonte caractere por caractere e extrai os *tokens*. A sequência de *tokens* gerada pelo analisador léxico é utilizada como entrada do analisador sintático, também chamado de *Parser*, conforme mostra a Figura 23. O *token* é a unidade básica que compõe o texto de um programa-fonte, que possui um significado na linguagem fonte. Exemplos: palavras reservadas, identificadores, constantes e operadores da linguagem.

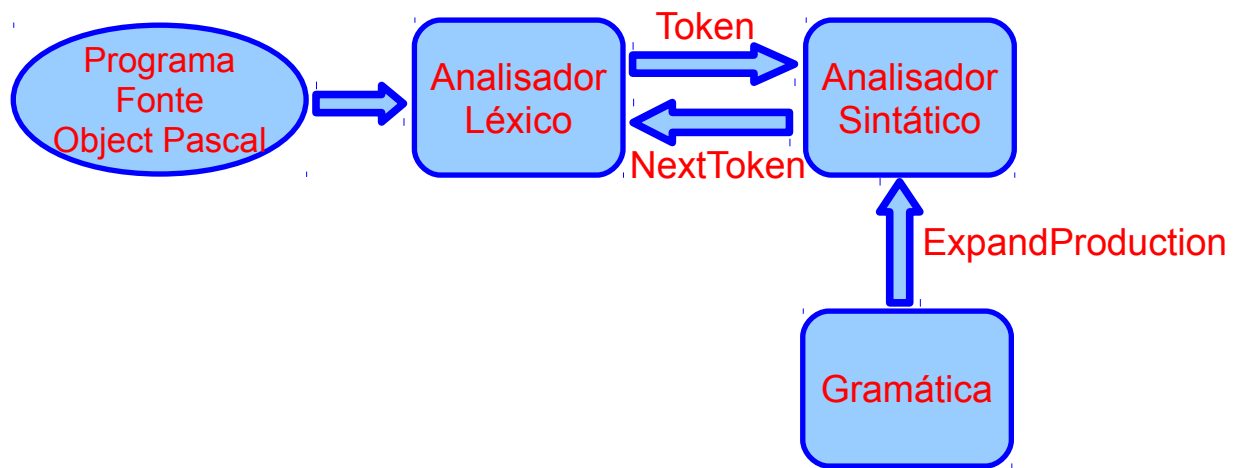


Figura 23 - Interface entre o analisador léxico e o sintático

Durante o processo de análise léxica são desprezados, do programa-fonte, caracteres não significativos, tais como espaços em branco, tabulações, caracteres de avanço de linha e comentários. Também são indicados os erros léxicos, tais como caracteres inválidos no programa-fonte, por exemplo letras com acento, fora de *strings*, que não aceitos em Pascal, programas-fonte que não foram achados pelo compilador e comentários ou *strings* não fechados. Ele também correlaciona a mensagem de erro com o nome do programa-fonte, número da linha e o número da coluna onde os erros léxicos, sintáticos e semânticos ocorrem.

Um exemplo de mensagem de erro fornecido pelo compilador é mostrado na Figura 24.

```
[Error] Unit1.pas (100, 20): E17 "INTERFACE" not found, but
"type" found
```

Figura 24 – Exemplo de mensagem de erro

Onde entre colchetes aparece a severidade da mensagem, que pode ser:

- a) **Hint** para dicas que podem melhorar o código, por exemplo indicação de variáveis declaradas mas não usadas, normalmente fornecidas pelo analisador semântico;
- b) **Warning** para erros de menor gravidade que não impedem a geração do código objeto, por exemplo erros de sintaxe em diretivas de compilação;
- c) **Error** para erros que impedem a geração do código objeto, por exemplo uma string não fechada com apóstrofes.
- d) **Fatal** para erros que impedem o prosseguimento da compilação do programa-fonte em análise.

Em seguida aparece o nome do programa-fonte Pascal e entre parêntesis o número da linha e o número da coluna onde o erro ocorreu dentro do programa-fonte. Por fim o código da mensagem e o texto da mensagem.

Como Object Pascal suporta macros preprocessadas, o *Scanner* é responsável por expandi-las à medida que a análise léxica se desenvolve, como exemplificado na Figura 25.

```
{ $DEFINE Fat = Fatorial(A); }
X := Fat
```

Figura 25 – Uma macro em Object Pascal

Nesse trecho o analisador léxico é instruído para sempre que encontrar a sequência de caracteres “Fat”, substituí-la por “Fatorial(A);” antes de fazer seu processamento.

3.1 SEPARAÇÃO DO ANALISADOR LÉXICO DO SINTÁTICO

Há várias razões para separar o analisador léxico do sintático, conforme mostra (FREITAS, 2008), entre elas:

- a) Simplificação de ambas as fases, através da especialização de cada uma, tornando

mais claro e modular o projeto do compilador.

- b) A performance do compilador pode ser substancialmente melhorada. Uma thread específica pode ser atribuída ao *Scanner*, pois a análise léxica é um processo I/O *bound*, onde muito tempo é gasto lendo programas-fonte do disco, e as análises sintática e semântica são CPU *bound*. Nas atuais arquiteturas *multicore* e *multithread* essa melhoria de performance torna-se clara.

3.2 O TOKEN

O *token* é a menor unidade que compõe um programa-fonte e que possui um significado específico na linguagem fonte. No LLVM-Pascal o *token* é representado pela classe TToken que é apresentada na Figura 26.

```
TToken = class
    Lexeme          : string;
    Kind            : TTokenKind;
    IntegerValue    : Int64;
    RealValue       : Extended;
    StringValue     : string;
    Hash            : Cardinal;
    Type_           : TToken;
    Scope           : Word;
    NextScope       : TToken;
end;
```

Figura 26 – Classe TToken

A classe TToken possui atributos que são criados, utilizados e modificados durante as quatro fases do frontend:

- a) **Lexeme**: O lexema é a cadeia de caracteres encontrada no programa-fonte que nomeia o *token*. Ele é identificado na análise léxica.
- b) **Kind**: É o tipo do *token*, implementado como uma enumeração, a cada fase do compilador este tipo pode ser alterado tornando-se mais específico. Por exemplo

na fase léxica o *token* pode ser reconhecido como um “Identificador” (tkIdentifier), posteriormente o analisador semântico pode promovê-lo para “Inteiro” (tkInteger), ou seja, de um identificador genérico para uma variável, mais específica, do tipo inteiro.

- c) **IntegerValue**, **RealValue** e **StringValue**: São atributos preenchidos pelo analisador léxico quando o *token* é reconhecido como uma constante ou literal representando um valor inteiro, real ou cadeia de caracteres respectivamente. Esse valor é utilizado, posteriormente, pelo gerador de código para gerar instruções que usarão essas constantes.
- d) **Hash**: É o código *hash* gerado quando o *token* é inserido na tabela de símbolos. O *hash* é usado como chave de acesso direto na tabela de símbolos. Isso ocorre quando o analisador sintático sinaliza, através da gramática, que o *token* está sendo usado para declarar uma variável ou um tipo.
- e) **Type_**: Atribuído e usado pelo analisador semântico para ligar *tokens* identificados como variáveis ou tipos a *tokens* identificados como tipos, isso permite a checagem de tipos feita pelo analisador semântico.
- f) **Scope**: Atributo mantido pelo analisador semântico para rastrear o escopo estático de um *token* classificado como variável ou tipo.
- g) **NextScope**: Atributo controlado pelo analisador semântico para aninhar ou desaninhar escopos estáticos usando a mesma tabela de símbolos. Para maiores detalhes veja o Capítulo 5.

3.3 EXPRESSÕES REGULARES

Como a análise léxica requer o reconhecimento de padrões para a identificação dos *tokens*, o uso de expressões regulares encaixa-se, perfeitamente, como principal técnica usada para sua implementação. A expressão regular, ou Regex, é uma notação padrão e compacta capaz de representar o reconhecimento de cadeias de caracteres (LOUDEN, 2004), essa notação se constitui numa linguagem formal que pode ser interpretada por um *engine* ou um método. No *Scanner* foi programado um método especial, `ScanChars`, capaz de interpretar um subconjunto da Regex e que é utilizado extensivamente no *Scanner*, simplificando sua

implementação. Por exemplo, para que um identificador seja reconhecido pelo *Scanner* a Regex seria expressa conforme mostra a Figura 27 com o correspondente código Pascal na linha seguinte.

```
(_A-Za-z)(_A-Za-z0-9)*
ScanChar([['_','A'..'Z','a'..'z'],['_','A'..'Z','a'..'z',
'0'..'9']], [1, 254]);
```

Figura 27 – Regex para reconhecer um identificador e o código correspondente

A semelhança e a relação direta entre a Regex e o código Pascal tornaram o analisador léxico mais legível, compacto e fácil de manter.

Dentro do *Scanner* o principal método, `NextToken`, basicamente é um *loop* que descarta brancos, comentários e que cria o objeto `Token` da classe `TToken`. No seu cerne há um grande comando `case` que invoca `ScanChars` que utiliza expressões regulares para o reconhecimento dos *tokens* retornando-os para o analisador sintático.

4 ANALISADOR SINTÁTICO

Esta é a segunda fase de um compilador e no caso do LLVM-Pascal é a mais importante, conforme será apresentado no decorrer deste capítulo. Nesta fase, também chamada de *Parser*, é verificado se a gramática da linguagem foi obedecida pelo programa-fonte. Uma árvore de derivação (AST – Abstract Syntax Tree) é gerada como efeito colateral dessa verificação. Com base na AST o analisador semântico faz mais análises e posteriormente o gerador de código pode criar o código intermediário. Esta interação entre as fases do compilador é mostrada na Figura 28.

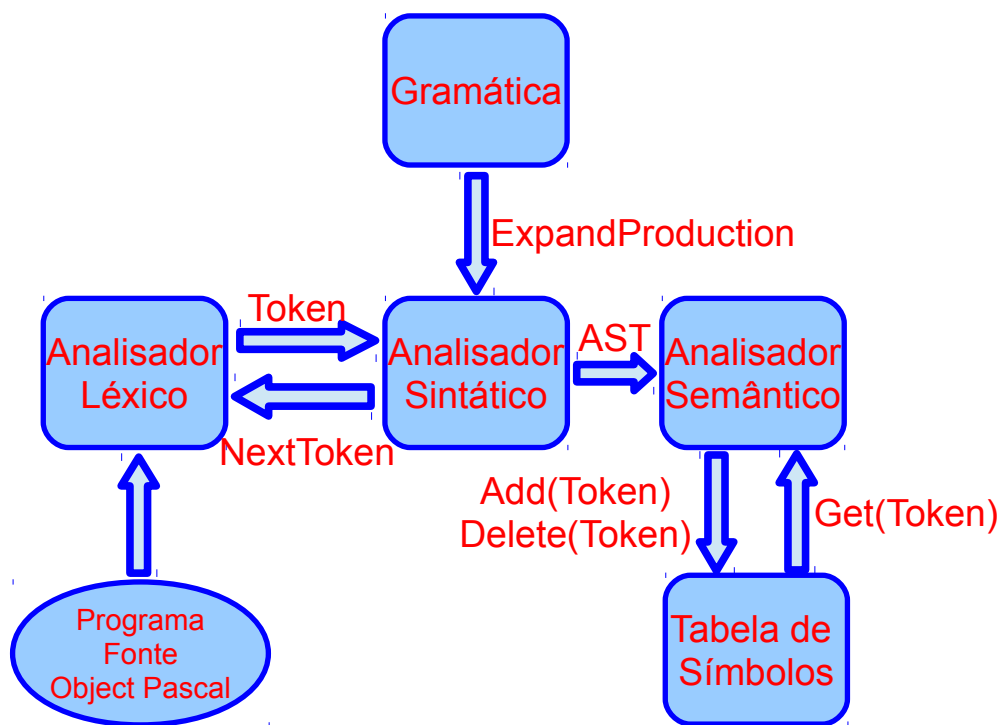


Figura 28 - Interação do analisador sintático com as demais fases do compilador

Neste capítulo são apresentadas as diversas técnicas utilizadas pelos analisadores sintáticos e indicaremos quais técnicas foram utilizadas pelo LLVM-Pascal e que modificações em tais técnicas foram feitas de forma a alcançarmos nosso objetivo de simplicidade e controle da complexidade do compilador.

4.1 GRAMÁTICA

A sintaxe das construções válidas em uma linguagem de programação pode ser precisamente especificada usando uma gramática livre de contexto, GLC ou simplesmente gramática. Uma gramática é um conjunto de regras, onde cada regra está na forma:

$$V_n \rightarrow W$$

A regra também é chamada de produção. O termo V_n é chamado de variável não terminal, símbolo não terminal ou simplesmente não-terminal. W é uma sequência de variáveis não terminais (V_n) e/ou variáveis terminais (V_t), que são os *tokens*, já descritos na Seção 3.2.

Por exemplo, em Object Pascal, um programa é formado por comandos compostos (blocos BEGIN/END), um comando composto é formado por comandos, um comando como o IF pode conter expressões, que são formadas, finalmente, por identificadores. Para representar essa hierarquia de composições, expressa pela gramática, convencionou-se usar a notação BNF (Backus-Naur Form). O que foi dito em linguagem natural neste parágrafo pode ser escrito em BNF, conforme a Figura 29.

```
<start> ::= PROGRAM <identificador>; <comandocomposto>.
<comandocomposto> ::= BEGIN <comando> END
<comando> ::= IF <expressão> THEN <comando>; | <comandocomposto>;
<expressão> ::= <identificador> [<operador> <identificador>]
```

Figura 29 – BNF do parágrafo anterior

A gramática completa do Object Pascal em BNF, que é implementada no *Parser* do LLVM-Pascal, pode ser encontrada no apêndice A.

4.2 COMPLEXIDADE

Na maioria dos parsers cada produção da gramática é implementada como uma *procedure* ou método. Numa linguagem como Object Pascal mais de 120 *procedures* são necessárias para a implementação de um *parser*. Essas mais de 120 *procedures* chamam umas às outras, inclusive de forma direta e indiretamente recursiva, tornando a construção, a manutenção e a depuração do *parser* tarefas de alta complexidade.

Para facilitar a construção de *scanners* e *parsers* há os compiladores de compiladores, tais como LEX/YACC (Yet Another Compiler to Compiler), que a partir de uma especificação, similar à BNF, são capazes de gerar os programas-fonte desses analisadores. Normalmente essa técnica é usada para facilitar a criação da primeira versão do compilador, que posteriormente terá sua evolução e manutenção realizadas de forma manual, resolvendo parcialmente a questão do domínio da complexidade de um compilador.

No caso do LLVM-Pascal, a complexidade do compilador é dominada tornando a gramática parte do compilador e não uma mera especificação de projeto. Para tanto foram escolhidas algumas técnicas específicas de compilação para tornar isso possível.

4.3 TIPOS DE ANALISADORES SINTÁTICOS

Primeiramente foi adotada a análise sintática descendente não recursiva LL(1) ao invés de análise sintática ascendente, tais como LR(1) e suas variantes (LR(k), SLR e LALR). Ambas as análises LL e LR servem para reconhecimento de gramáticas livres de contexto.

4.3.1 Analisador Descendente

A análise descendente ou *top-down* (LL, Left-Left) lê a entrada da esquerda para a direita, daí o primeiro “L”. Começa o reconhecimento da linguagem a partir da produção mais genérica da gramática, nó raiz ou símbolo inicial, conhecido como `<start>`. O reconhecimento desce até os *tokens* e continua subindo e descendo pelas produções intermediárias montando uma árvore, chamada árvore de derivação ou AST (Abstract Syntax Tree). Esse reconhecimento produz uma derivação mais à esquerda da árvore de derivação ou

percorrimento pós-ordem, conforme mostra a Figura 30, daí o segundo “L” de LL (Left-Left).

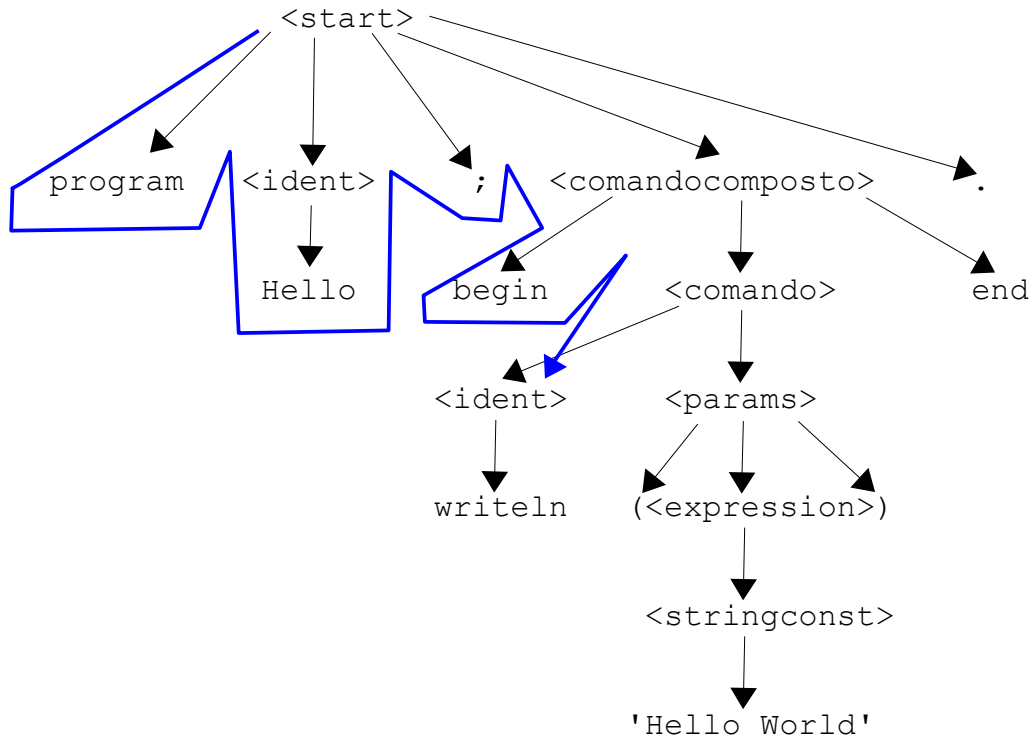


Figura 30 – Análise descendente para um programa Hello World

O “k” na notação LL(k) refere-se ao número de *tokens* que precisam ser conhecidos além do *token* atual para que o *parser* possa continuar seu reconhecimento. Quanto maior for k mais complexa é a gramática a ser analisada. O *parser* associado também será mais complexo e exigirá maiores recursos computacionais. Quando $k = 1$, k normalmente é omitido, ou seja, $LL(1) = LL$. Por sua simplicidade e pela maior facilidade em ser implementado à mão, o *parser* adotado no LLVM-Pascal é do tipo LL(1). Os *parsers* preditivos (um tipo de descendente recursivo) não precisam de retrocesso, ou seja, não precisam reler a entrada. A maioria das linguagens de programação atuais, como Object Pascal, não precisam de retrocesso e podem ser expressas em gramáticas do tipo LL(1), que são reconhecidas por *parsers* LL(1).

Para tornar o *parser* ainda mais simples e flexível ele é “não recursivo”, pois implementa a pilha de produções de forma explícita e não de forma implícita, que seria através da chamada recursiva de *procedures* que implementam as produções. Com isso

eliminou-se mais 120 *procedures* que foram substituídas por 5 métodos que implementam uma máquina virtual de pilha de produções, que lê e interpreta a gramática do LLVM-Pascal diretamente. Essa máquina virtual, que é o próprio *parser*, é descrita na Seção 4.7.

4.3.2 Analisador Ascendente

A análise ascendente ou *bottom-up* (LR, Left-Right) lê a entrada da esquerda para a direita, daí o primeiro “L”, mas começa o reconhecimento a partir das produções mais específicas da gramática, ou seja, cria a árvore de derivação a partir das folhas seguindo até o símbolo inicial da gramática, no caso `<start>`. Se o *parser* consegue chegar ao símbolo inicial então o reconhecimento da linguagem foi bem sucedido. Este processo de substituição de produções mais específicas por outra produção mais genérica é chamado de redução.

Dentre as dificuldades em se implementar um analisador LR estão:

- a) A determinação da subcadeia de *tokens* que deverá ser reduzida.
- b) A escolha da produção que fará a redução, uma vez que mais de uma escolha pode ser possível.

Por ter uma implementação mais complexa, os analisadores LR são normalmente gerados a partir de compiladores de compiladores, tais como o YACC. Como tal procedimento gera muito código, mesmo que automaticamente, optou-se pelo descarte dessa abordagem na construção do *parser* para o compilador.

4.4 TERMINAIS FIRST/FOLLOW

A implementação de *parsers* LL é bastante facilitada usando o conceito de terminais FIRST/FOLLOW (AHO, 2008). Esses terminais permitem escolher diretamente que produção deverá ser aplicada baseando-se no *token* sob análise, isso simplifica o *parser* e aumenta sua performance.

FIRST(A) é definido como o conjunto de símbolos terminais que podem começar o símbolo não terminal A. Por exemplo, seja a gramática, conforme mostra a Figura 31.

```

<Start> ::= <Program>|<Unit>|<Library>|<Package>
<Program> ::= PROGRAM<Ident>;<DeclSection><CompoundStmt>.
<Unit> ::= UNIT<Ident>;<IntSection><ImplSection><InitSection>.
<Library> ::= LIBRARY<Ident>;<UsesClause><InitSection>.
<Package> ::= PACKAGE<Ident>;<Requires><Contains>END.

```

Figura 31 – Gramática de exemplo para FIRST(<Start>)

Então o conjunto de terminais FIRST para o não terminal <Start> da gramática do Object Pascal são: PROGRAM, UNIT, LIBRARY e PACKAGE, o que é expresso formalmente como:

$$\text{FIRST}(\langle \text{Start} \rangle) = \{ \text{'PROGRAM'}, \text{'UNIT'}, \text{'LIBRARY'}, \text{'PACKAGE'} \}$$

Ou seja, um programa-fonte em Object Pascal deve começar com uma dessas quatro palavras reservadas, definindo respectivamente um programa, unidade, biblioteca ou pacote. Cada um desses terminais encabeçam uma produção que é imediatamente acionada pelo parser quando detecta um desses quatro *tokens* quando estiver analisando o símbolo não terminal <Start>.

Por sua vez FOLLOW(A) representa o conjunto de terminais que ocorrem à direita de um símbolo não terminal A em uma dada produção. Considerando a gramática da Figura 31, temos:

$$\text{FOLLOW}(\langle \text{Contains} \rangle) = \{ \text{'END'} \}$$

Por ser um conceito subsidiário e pela simplicidade do LLVM-Pascal, os terminais FOLLOW não precisaram ser usados no *Parser*.

4.5 RECUPERAÇÃO DE ERROS COM FIRST

Durante a recuperação de erros os terminais FIRST/FOLLOW podem ser usados como *tokens* de sincronismo do *parser* de forma que ele continue a avaliar a cadeia de entrada mostrando apenas erros significativos e isolados dos erros anteriores. Na recuperação de erros do LLVM-Pascal é usado o modo pânico, mas com um algoritmo mais elaborado do que o indicado na literatura (AHO, 2008). O *token*, em análise, é comparado com todos os terminais

FIRST válidos considerando a pilha de produções ativa e sua ordem inversa de ativação. Se alguma produção tem o mesmo terminal FIRST, esta produção é escolhida para continuar a análise. Se não o *Scanner* é acionado para trazer o próximo *token* e o procedimento de recuperação é repetido até encontrar uma produção adequada ou o fim do programa-fonte.

4.6 SDT - SYNTAX DIRECTED TRANSLATION

Uma notação complementar usada no projeto de um analisador sintático é a SDT (Syntax Directed Translation Scheme, Esquema de Tradução Dirigido pela Sintaxe) (AHO, 2008). Uma SDT é uma gramática livre de contexto com ações semânticas, que podem aparecer em qualquer ponto no corpo de uma produção. Uma ação semântica nada mais é que o nome de um método do analisador semântico a ser invocado pelo analisador sintático assim que todos os símbolos da produção, em análise, à esquerda da ação semântica tiverem sido validados.

4.6.1 SDTF – SDT with FIRSTs

No LLVM-Pascal a gramática foi escrita na forma de SDT, refatorada usando terminais FIRST, incorporada diretamente ao programa-fonte e interpretada em tempo de execução por uma máquina virtual de pilha.

A gramática em BNF da Figura 31, convertida para SDT refatorada com terminais FIRST é mostrada na Figura 32.

```
// Start
    '{PROGRAM}' + PushScope + Ident + AddModule + ';' +
DeclSection + CompoundStmt + '.' + PopScope +
    '{UNIT}' + PushScope + Ident + AddModule + ';' +
IntSection + ImplSection + InitSection + '.' + PopScope +
    '{LIBRARY}' + PushScope + Ident + AddModule + ';' +
UsesClause + InitSection + '.' + PopScope +
    '{PACKAGE}' + PushScope + Ident + AddModule + ';' +
Requires + Contains + 'END.' + PopScope,
```

Figura 32 – BNF transformada em SDT refatorada com terminais FIRST

Vamos chamar esta nova notação de SDTF. Em SDTF o nome da produção `Start` é definida como:

```
const
    Start = Syntatic + #000;
```

`Start` é uma constante *string* de dois caracteres, o primeiro indica que é um símbolo a ser tratado pelo analisador sintático e o segundo é um sequencial começando com 0, usado como indexador de um *array* de SDTFs. No *array* de SDTFs o primeiro elemento é a produção `Start`, conforme mostrado na Figura 32. Em SDTF todos os terminais são escritos em letras maiúsculas e dentro de apóstrofes, também chamadas de aspas simples. Os terminais `FIRST`, além disso, ficam entre chaves. Os símbolos não terminais ou produções (por exemplo `Ident` e `DeclSection`) e as ações semânticas (por exemplo `PushScope`, `PopScope` e `AddModule`) são escritos com a regra de capitalização do `PascalCase` (similar ao `camelCase`, mas com a primeira letra em maiúscula). A produção começa com um terminal `FIRST`, cada elemento da produção é separado pelo operador `+` e termina com uma vírgula. SDTF adota essa forma, porque é uma *string* Object Pascal dentro de um *array* de *strings*.

4.7 MÁQUINA VIRTUAL DE PILHA

Para interpretar um programa-fonte o algoritmo do *Parser* usa o conceito de máquina virtual de pilha. Essa máquina usa uma gramática SDTF como seu código objeto, conforme algoritmo mostrado na Figura 33.

```
Empilhar Start;
while not Fim de Programa begin
    Simbolo ← Topo da Pilha;
    case Tipo de Simbolo of
        Não Terminal: Expandir Produção na Pilha;
        Terminal: Comparar com NextToken;
        Ação Semântica: Chamar método do analisador semântico;
        Ação Geradora: Chamar método do gerador;
    end;
```

```

Desempilhar;
end;

```

Figura 33 – Algoritmo da máquina virtual de pilha

A máquina de pilha começa empilhando o símbolo *Start*, em seguida avalia o tipo do símbolo no topo da pilha. Como *Start* é um símbolo não terminal é solicitada sua expansão na pilha. O procedimento de expansão na pilha (*ExpandProduction*) invoca o *NextToken* e acessa a produção *Start* na gramática. Dentro da produção *Start* procura um *FIRST* idêntico ao *NextToken*, se achar expande essa produção na pilha em ordem inversa, ou seja, da direita para a esquerda. Retoma o *loop* principal e avalia o tipo do símbolo na pilha. No caso o terminal 'PROGRAM' aparece no topo da pilha, veja Figura 34. É feito o reconhecimento, o símbolo é desempilhado e o algoritmo prossegue. Quando uma ação semântica (por exemplo *PushScope*) ou uma ação de geração são encontradas, o método correspondente no analisador semântico ou gerador de código é invocado, o símbolo é desempilhado e o algoritmo continua até o fim do programa-fonte. Se um erro é encontrado, uma mensagem de erro é mostrada, conforme mostrado na Figura 24, e é executada a recuperação de erros descrita na Seção 4.5.

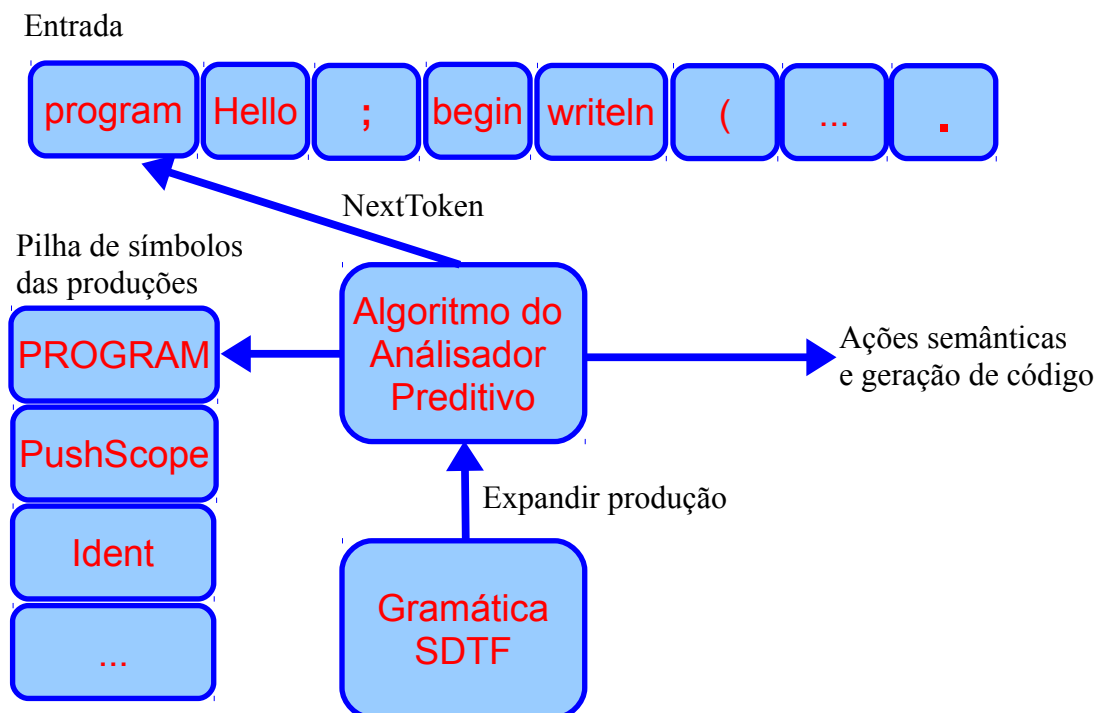


Figura 34 – Máquina virtual de pilha alimentada pela gramática

5 ANALISADOR SEMÂNTICO

O analisador semântico, também chamado *Analyser*, *Checker* ou *Type-Checker*, é a terceira fase do compilador. Ele faz o processo de verificações estáticas, que são consistências semânticas feitas durante a compilação.

As verificações estáticas têm grande importância para que o LLVM-Pascal possa fazer uma compilação bem sucedida, com o mínimo de erros, pois elas capturam erros adicionais que o *Lexer* e o *Parser* não são capazes de detectar.

As verificações estáticas consistem em restrições adicionais à gramática, permitindo:

- a) Verificar se os tipos dos operandos são compatíveis dentro de expressões aritméticas e booleanas;
- b) Verificar se o número e os tipos dos parâmetros nas chamadas de funções e métodos foram respeitados;
- c) Fazer a coerção de tipos (alargamento de tipos) dentro de expressões aritméticas, quando necessária. Por exemplo numa soma ou numa comparação entre uma variável real e outra inteira, a variável inteira precisa ser convertida ou alargada para real antes de ser somada ou comparada com a variável real. Esta conversão ou alargamento é chamado de coerção. O manual de referência do Object Pascal (CANNYET, 2011) define quais coerções são permitidas pela linguagem;
- d) Verificar se um identificador foi declarado no máximo uma vez em um escopo;
- e) Verificar se um identificador não declarado está sendo usado;
- f) Verificar se um comando `break` ou `continue` está dentro de um *loop*;
- g) Verificar se um identificador declarado não foi usado no seu escopo;
- h) Verificar se *units* declaradas não foram usadas no escopo atual;
- i) Verificar se literais constantes ou parâmetros constantes estão sendo atribuídos;
- j) Verificar se o número e os tipos dos indexadores de *arrays* são foram respeitados;

Para realizar boa parte dessas verificações o analisador semântico precisa distinguir entre os identificadores que aparecem à esquerda (valor-l) dos que aparecem à direita (valor-r) de uma atribuição. Isso é facilmente obtido através das ASTs que são geradas durante o processo de *parsing* e das ações semânticas disparadas pelas SDTFs.

Diferentemente do que prega a literatura (AHO, 2008), que sugere o analisador sintático, o analisador semântico do LLVM-Pascal é quem implementa a tabela de símbolos, ou seja, é ele quem inclui, exclui, consulta e faz varreduras na tabela de símbolos para fazer suas verificações estáticas e repassa os *tokens* da tabela de símbolos para o gerador de código intermediário. Pois, pela arquitetura do LLVM-Pascal, é ele quem está na melhor posição para distinguir por exemplo entre uma declaração e o uso de um determinado identificador.

5.1 TABELAS DE SÍMBOLOS

As *Symbol Tables* (ST) são estruturas de dados usadas pelos compiladores para guardar informações sobre as construções do programa fonte. As entradas na tabela de símbolos são criadas e usadas durante a fase de análise semântica e do gerador de código intermediário. As informações da ST são coletadas e descartadas ao se realizar o encaminhamento pós-ordem da AST, discutido na Seção 5.3.1, e usadas pelo gerador de código para gerar as instruções de três operandos. As entradas na tabela de símbolos são objetos da classe TToken, já especificada na Seção 4.2.

As tabelas de símbolos precisam dar suporte ao conceito de Escopo Estático em linguagens como Object Pascal, como mostra a Figura 35.

```

program Escopo1;
var
    W1, X1, Y1 : integer;
procedure Escopo2;
var
    W2, Y2, Z2 : integer;
begin
    writeln(W2, X1, Y2, Z2);
end;
begin
    writeln(W1, X1, Y1);
end.

```

Figura 35 – Escopo estático em um programa Pascal

No escopo estático pode haver múltiplas declarações aninhadas em blocos ou procedimentos do mesmo identificador dentro de um programa. No exemplo da Figura 35, as variáveis *W* e *Y* estão declaradas repetidamente no Escopo1 e no Escopo2, observe o uso de subscritos para distinguir entre as declarações de um mesmo identificador. Para resolver essa situação AHO (AHO, 2008) sugere criar STs separadas e aninhadas para cada escopo.

5.2 DIFERENCIAL DA TABELA DE SÍMBOLOS NO LLVM-PASCAL

Um diferencial do LLVM-Pascal é que ele usa uma única ST com um algoritmo *Hash*, que trata o escopo automaticamente sem a necessidade do aninhamento de outras STs.

Essa ST única mantém acessíveis apenas as entradas das variáveis do escopo corrente. Essa tabela *Hash* admite basicamente pesquisas de tempo constante $O(1)$, à custa da inserção e exclusão de entradas na entrada e saída do escopo, as ações semânticas *PushScope* e *PopScope*.

No exemplo da Figura 35, no Escopo1 as variáveis são inseridas normalmente na ST e numa pilha auxiliar, como mostra o lado esquerdo da Figura 36. Na entrada do Escopo2 (lado direito da Figura 36) as variáveis ocupam o lugar das variáveis homônimas do Escopo1 na ST e mantêm um ponteiro para elas, além disso as variáveis do Escopo2 são colocadas na pilha auxiliar sobre as variáveis do Escopo1.

Por exemplo: Quando a variável W_2 é inserida na ST, ela ocupa o lugar de W_1 , W_2 mantém um ponteiro para W_1 e W_2 é inserida na pilha numa posição acima de W_1 . Neste momento quando o *Analyser* procura *W* na ST, por causa da referência a *W* no comando `writeln(W, ...)`, irá achar W_2 e não W_1 . Este aninhamento garante a implementação do escopo estático em Pascal. No topo da pilha estão as variáveis para o escopo corrente. Abaixo delas, na pilha, estão as variáveis para os escopos envolventes. Assim, os escopos podem ser alocados e liberados em um padrão tipo pilha.

Após a saída do Escopo2, ou seja, voltando para o Escopo1 (voltando para o lado esquerdo da Figura 36), o *Analyser* precisa desfazer quaisquer mudanças ocorridas na ST devido às declarações do Escopo2. Ele faz isso desempilhando as variáveis do Escopo2 e reposicionando as variáveis do Escopo1 de volta às suas posições originais na ST.

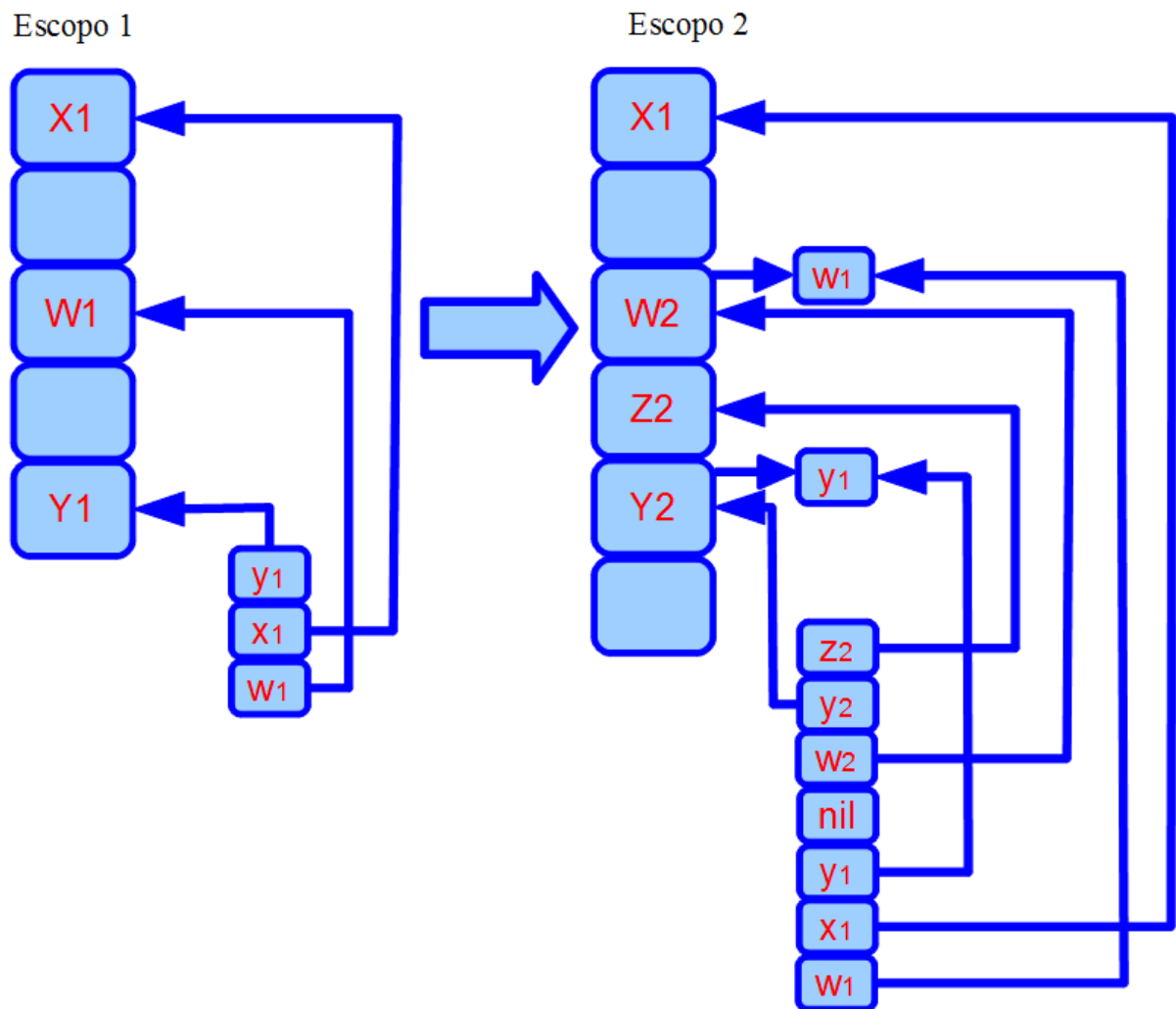


Figura 36 – Configuração da Tabela de Símbolos ao aninhar o Escopo2 no Escopo1

6 GERADOR DE CÓDIGO INTERMEDIÁRIO

O gerador de código intermediário, ou *Generator*, é quarta fase do compilador, ele é responsável pelo processo pegar as ASTs e a ST e transformar em um código de três endereços que é a LLVM-IR (Intermediate Representation). O gerador de código intermediário pode produzir vários tipos de representações intermediárias. Neste trabalho é usada a forma intermediária chamada código de três endereços, que consiste em uma sequência de instruções do tipo *assembly* do LLVM com três operandos por instrução. Cada operando pode atuar como um registrador. A saída do gerador de código intermediário consiste em uma sequência de instruções de três endereços.

Durante o *parsing* as ASTs são montadas e as SDTFs são interpretadas executando-se as ações semânticas, nelas declaradas, para gerar o código de três endereços.

6.1 CÓDIGO DE TRÊS ENDEREÇOS

O código de três endereços é uma instrução *assembly* na forma:

$$X = Y \text{ op } Z$$

Onde x , y , z são variáveis ou registradores emitidos pelo *Generator* e op é um operador binário.

Para cópia de valores é aceita a forma:

$$X = Y$$

No caso de *arrays* são usadas duas variações do código de três endereços, conforme mostra a Figura 37.

$$X[Y] = Z$$

$$X = Y[Z]$$

Figura 37 – Código de três endereços para *arrays*

A primeira instrução atribui Z para $X[Y]$ e a segunda atribui $Y[Z]$ para X .

As instruções de três endereços são executadas na ordem em que foram emitidas pelo

Generator. Como em outros *assemblies*, essa ordem só pode ser mudada por instruções de desvio condicional ou incondicional, como as mostradas na Figura 38.

- 1) `ifFalse x goto L`
- 2) `ifTrue x goto L`
- 3) `goto L`

Figura 38 – Instruções de controle de fluxo para código de três endereços

As instruções de controle de fluxo da Figura 38 funcionam assim:

- 1) Se `x` é falso execute a instrução com rótulo `L`.
- 2) Se `x` é verdadeiro execute a instrução com rótulo `L`.
- 3) Vá para a instrução com rótulo `L`.

Um rótulo `L` pode ser conectado a qualquer instrução iniciando-a com um prefixo `L`:. Uma instrução pode ter mais de um rótulo.

6.2 TRADUÇÃO DE COMANDOS

Para traduzir o comando Pascal `if <expr> then <stmt>` para código de três endereços usa-se instruções de desvio, conforme mostrado na Figura 39.

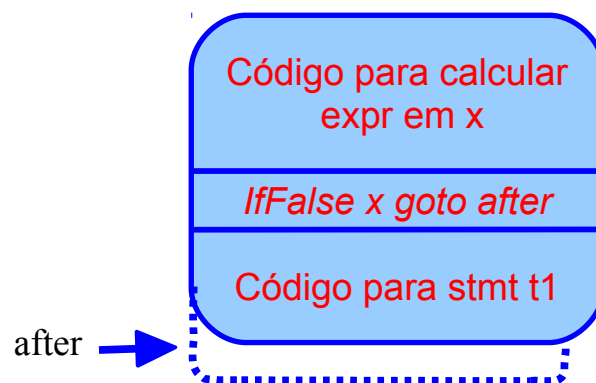


Figura 39 – Leiaute do código para o comando `if`

O comando de desvio salta para o rótulo `after` se `expr` for avaliado como `false`. Outros comandos tais como `while`, `repeat`, `if-then-else` podem ser traduzidos de forma similar usando instruções de desvio.

6.3 TRADUÇÃO DE EXPRESSÕES

A tradução de expressões considera as expressões contendo operadores binários, acesso a *arrays*, atribuições e operandos na forma de constantes e identificadores.

Para cada nó operador na AST, que pertence a uma expressão, gera-se uma instrução de três endereços. Constantes e identificadores não geram, pois eles são operadores nas instruções de três endereços. Se um nó em um não terminal `Expr` tiver um operador, então uma instrução é emitida para calcular o valor no nó para um nome temporário gerado pelo *Generator*, em LLVM-IR os temporários começam com '%' seguido por um sequencial. Assim a expressão $A + B * (C + 4)$ é traduzida para três instruções. Ver Figura 40.

```
%1 = C + 4
%2 = B * %1
%3 = A + %2
```

Figura 40 - Tradução de uma expressão usando instrução de três endereços

No caso de *arrays* e atribuições, deve-se levar em consideração a diferença entre os valores-l e os valores-r. Por exemplo para $A + B[C]$ deve ser calculado o valor-r de $B[C]$ em um temporário, como mostrado na Figura 41.

```
%4 = B[C]
%5 = A * %4
```

Figura 41 - Expressões usando *arrays*

Para um exemplo mais completo, seja $B[C] = A + B[D*4]$. A geração de código é mostrada na Figura 42.

```
%6 = D * 4
%7 = B[%6]
%8 = A + %7
B[C] = %8
```

Figura 42 – Exemplo mais completo com *arrays*

7 MICRO COMPILADOR DE EXPRESSÃO ARITMÉTICA

O objetivo deste micro compilador é o de exemplificar de forma simples um compilador e algumas de suas principais fases, a saber a análise léxica, a análise sintática, a análise semântica e a geração de código intermediário. Ele foi escrito em Pascal para servir de modelo em pequena escala de como será o compilador final (FREIRE, 2011).

7.1 A LINGUAGEM FONTE

A linguagem fonte é uma das mais conhecidas possível, a saber a tradicional linguagem de expressão aritmética, conforme mostra a Figura 43.

```

<Expressao_Aritmetica_Simples> ::=
E   ::= T E'
E'  ::= + T E' | e
T   ::= F T'
T'  ::= * F T' | e
F   ::= ( E ) | constante

```

Figura 43 – BNF para expressão aritmética

O símbolo inicial da gramática é a variável não terminal E. Essa linguagem não possui identificadores e em função disso não possui tabela de símbolos.

7.2 ANÁLISE LÉXICA

O analisador léxico, em sua essência, ocupa apenas duas linhas de fonte marcadas na cor **vermelha**. Conforme apresentado na Seção 7.5 a seguir.

7.3 ANÁLISE SINTÁTICA

O analisador sintático foi construído segundo o método descendente, técnica que também é usada no compilador definitivo. O analisador sintático está destacado na cor **rosa**. Observe que além das rotinas de análise descendente em si, também foram marcadas, as

rotinas auxiliares (Follow e Eat).

7.4 ANÁLISE SEMÂNTICA E GERAÇÃO DE CÓDIGO

O analisador semântico e a geração de código intermediário formam um corpo só, aqui marcados na cor **verde**. O algoritmo utilizado transforma expressões aritméticas infixadas em notação polonesa pós-fixada. As rotinas semânticas são inseridas na BNF como variáveis não terminais comuns e se transformam em procedimentos, essa técnica também será aproveitada no compilador final. A BNF estendida com a semântica fica definida conforme mostra a Figura 44.

```

<Expressao_Aritmetica_Simples_com_Semantica> ::=
E  ::= T E' S4
E'  ::= S5 + T E' | e
T   ::= F T'
T'  ::= S5 * F T' | e
F   ::= S3 ( E S2 ) | S1 constante

```

Figura 44 – BNF para expressão aritmética com ações semânticas

Essa gramática é a mesma da Figura 43, mas com a inserção das ações semânticas.

7.4.1 Forma infixada para notação polonesa

Segue-se o algoritmo de conversão da expressão aritmética na forma infixada para a notação polonesa. Ver Figura 45.

```

InicializaPilha;
repeat
    Lexico; // Lê o próximo token
    if (Token = ID) or (Token = CONSTANTE) then
        Write(Token)
    else
        if Token = ')' then begin
            while Pilha[Topo] <> '(' do

```

```

        Write(Desempilha);
        Desempilha; // Descarta
    end
else begin
    if (Token <> '(') and not pilha_vazia then
        while Precedencia(Pilha[Topo]) <=
            Precedencia(Token) do
            Write(Desempilha);
            Empilha(Token);
        end
    until Token = EOF;
    while not PilhaVazia do Write(Desempilha);

```

Figura 45 – Algoritmo: Forma infixada para notação polonesa

A Tabela 7 mostra a precedência de operadores:

Tabela 7 - Precedência dos operadores para o micro compilador

Precedência	Operador
1	* /
2	+ -
3	()

7.5 O MICRO COMPILADOR PARA EXPRESSÃO ARITMÉTICA

A Figura 46 delimita as fases da compilação. Convencionamos dar cores diferentes aos trechos do fonte, conforme legenda:

- a) **Azul**: Para comandos e convenções da linguagem Pascal;
- b) **Preto**: Para comandos auxiliares;
- c) **Vermelho**: para o analisador léxico;
- d) **Rosa**: Para o analisador sintático;
- e) **Verde**: Para a análise semântica e geração de código intermediário.

```

program MicroCompilador;
uses WinCrt;
var
    TkVal,           // Token value
    Tk      : char;   // Token
    ToS     : longint; // Top of Stack
    Pilha   : array[0..15] of char;
    Arq     : text;
procedure E; forward;
procedure Inicio; begin
    Write('> ');
    ToS := 0;
    Assign(Arq, 'CodigoGerado.dat'); Rewrite(Arq);
end;
procedure Final; begin
    Close(Arq); WriteLn(#13, 'Ok'); ReadKey;
end;
procedure Erro; begin
    WriteLn(#13, 'Erro' ); Final;
end;
procedure Lex; begin
    Tk := ReadKey;
    if Tk < #31 then Tk := '$';
    TkVal := byte(Tk) - 48;
    Write(Tk);
end;
procedure Eat(T : char); begin
    if Tk = T then
        Lex
    else
        Erro;

```

```

end;

function FlwEl : Boolean; begin
    FlwEl := Tk in [')', '$'];
end;

function FlwTl : Boolean; begin
    FlwTl := FlwEl or (Tk = '+');
end;

function Prcd(C : char) : integer; begin
    case C of
        '+', '-' : Prcd := 1;
        '*', '/' : Prcd := 2;
        '(', ')' : Prcd := 3;
    end;
end;

end;

procedure Emit(Opr : String; Opnd : Longint); begin
    WriteLn(Arq, Opr, Opnd);
end;

procedure EmitOpr(Opr : char); begin
    case Opr of
        '+' : WriteLn(Arq, 'ADD');
        '*' : WriteLn(Arq, 'MULT')
    else
        Erro;
    end;
end;

end;

procedure Empilha(C : char); begin
    inc(ToS); Pilha[ToS] := C;
end;

procedure S1; begin
    Emit('PUSH cte ', TkVal);
end;

```

```

procedure S2; begin
    while (Pilha[ToS] <> '(') and (ToS > 0) do begin
        EmitOpr(Pilha[ToS]); dec(ToS);
    end;
    if ToS >= 1 then dec(ToS);
end;

procedure S3; begin
    if (Tk <> '(') and (ToS <> 0) then
        while Prcd(Pilha[ToS]) <= Prcd(Tk) do begin
            EmitOpr(Pilha[ToS]); dec(Tos);
        end;
        Empilha(Tk)
end;

procedure S4; begin
    while ToS > 0 do begin
        EmitOpr(Pilha[ToS]); dec(ToS);
    end
end;

procedure S5; begin
    Empilha(Tk);
end;

procedure F; begin
    case Tk of
        '0'..'9' : begin S1; Eat(Tk); end;
        '('      : begin S3; Eat('('); E; S2; Eat(')'); end;
    else
        Erro;
    end;
end;

```



```

procedure T1; begin
    if Tk = '*' then begin
        S5; Lex; F; T1
    end
    else
        if FlwT1 then
            {vazio}
        else
            Erro;
    end;
procedure T; begin
    F; T1;
end;
procedure E1; begin
    if Tk = '+' then begin
        S5; Lex; T; E1
    end
    else
        if FlwE1 then
            {vazio}
        else
            Erro;
    end;
procedure E; begin
    T; E1;
end;
begin
    Inicio;
    Lex;
    E; S4;
    Final
end.

```

Figura 46 – Fonte do micro compilador

A Figura 47 mostra o micro compilador sendo executando e informando a expressão $(9+8) * (7+6)$:

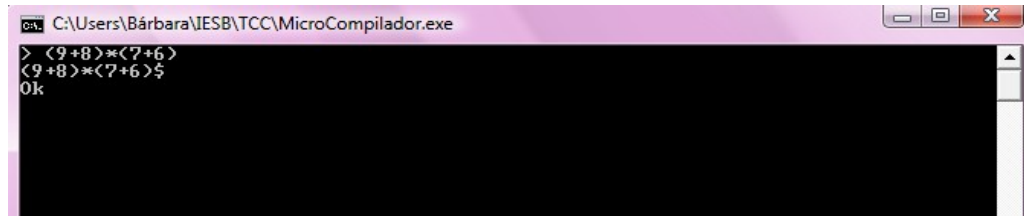


Figura 47 - Exemplo de compilação do micro compilador

Este exemplo gera o seguinte código intermediário em um arquivo texto com o nome: “CodigoGerado.dat”, conforme mostra a Figura 48.

```
PUSH cte 9
PUSH cte 8
ADD
PUSH cte 7
PUSH cte 6
ADD
MULT
```

Figura 48 - Código intermediário gerado pelo micro compilador

8 BACKEND LLVM

É um *framework* composto de um conjunto de bibliotecas reutilizáveis em C++ para a construção de compiladores, interpretadores e otimizadores de código. O LLVM começou como um projeto de pesquisa da Universidade de Illinois (Urbana-Champaign) em 2001. A partir de 2005 tornou-se um projeto estratégico da Apple, que o utiliza em três frentes principais:

- a) Para compilar seus sistemas operacionais: Mac OS X e iOS;
- b) Como base de seu ambiente de desenvolvimento Xcode e
- c) Para seu novo compilador para C/C++ e Objective C, o Clang, que como o LLVM-Pascal também é um *frontend*, mas para linguagens do tipo “C”. O Clang tem sido apresentado pela Apple como uma alternativa viável para a substituição do GCC (GNU C Compiler) com velocidade de compilação até três vezes maior, mensagens de erro mais expressivas, código gerado mais eficiente (em alguns cenários) e um esquema de licenciamento mais liberal.

Por ter uma licença BSD (Berkeley Software Distribution), bastante permissiva, o LLVM tem alcançado muita popularidade no mundo open source, com um envolvimento sempre crescente de colaboradores, pesquisadores, universidades, empresas e projetos. Com este desenvolvimento ativo o código vem sendo atualizado para as últimas tendências e tecnologias na área de arquiteturas de hardware e sistemas operacionais. Um novo *release* do *framework* é testado e liberado a cada 6 meses gerando um ritmo de atualização previsível e constante que ajuda no planejamento do *release* dos muitos projetos que estão sendo derivados do LLVM.

Com a utilização do LLVM neste projeto, mais da metade do esforço de construção do compilador pode ser eliminada. Bastando ao *frontend* gerar o LLVM-IR.

O LLVM-IR ou *bitcode* é a entrada padrão para várias fases e módulos do *Core* do LLVM. A partir dele o *Core* e os subprojetos do LLVM são capazes de realizar:

- a) Compilação estática ou tradicional;
- b) Compilação dinâmica ou JIT (Just in Time);
- c) Interpretação de código;

- d) Otimização de código dependente e independente de máquina;
- e) Otimização *off-line*, quando o código já está em produção;
- f) Verificação da correção lógica de programas;
- g) Tradução entre linguagens de alto nível, por exemplo de Pascal para C, e
- h) Instrumentação de código para detecção de erros de acesso à memória, que podem comprometer o funcionamento e a segurança do *software*, similar ao Valgrind (VALGRIND, 2012).

Além de subprojetos oficiais do LLVM, há uma ampla variedade de outros projetos que utilizam componentes de LLVM para várias tarefas. Através desses projetos o LLVM pode compilar Ruby, Python, Haskell, Java, D, PHP, Pure, Lua e o Object Pascal será a próxima linguagem deste rol, graças a este trabalho e várias outras linguagens. É por isso que está sendo usado para uma variedade tão grande de tarefas diferentes, desde seu uso em linguagens interpretadas como Lua até a compilação de programas científicos feitos em FORTRAN para rodar em supercomputadores massivamente paralelos.

8.1 LLVM – IR

É uma linguagem do tipo *assembly*, que possui atribuição estática e uma representação baseada em segurança de tipos, operações de baixo nível, que fornece a flexibilidade, e a capacidade de representar as linguagens de alto nível.

O LLVM-IR é a representação de código comum usado em todas as fases da estratégia de compilação do LLVM.

A representação de código LLVM é projetado para ser usada em três formas diferentes: em memória, representação *bitcode* em disco (adequada para carregamento rápido por um compilador *Just-In-Time*) e como um conjunto legível representação da linguagem. Essas três formas permitem que LLVM possa fornecer uma poderosa representação intermediária para transformações de compiladores eficientes e análise, fornecendo um meio natural para depurar e visualizar as transformações. Essas três formas diferentes de LLVM são equivalentes.

9 CONCLUSÃO

A proposta do Compilador LLVM-Pascal é a criação do primeiro compilador para linguagem Object Pascal usando o backend LLVM da Apple, com funcionalidades similares a um compilador profissional. Porém mais simples, com uma quantidade muito menor de linhas de código, mais performático, inteligível, de fácil manutenção, mais amigável para programadores, mostrando mensagens de erros efetivas e claras, que possa ser usado em aplicações científicas e comerciais do mundo real e ainda como material de estudo para fins didáticos. Para tanto o compilador está dividido em um *frontend* escrito em Object Pascal e em um *backend* escrito em C++, o LLVM da Apple, que foi completamente reutilizado.

Foi usado o Lazarus como ambiente de desenvolvimento para construção do LLVM-Pascal. Nele foram desenvolvidos e testados completamente o analisador léxico e o analisador sintático, que já seguem toda a especificação da linguagem Object Pascal (CANNYET, 2011). O analisador semântico e o gerador de código foram parcialmente desenvolvidos e ainda não implementam os paradigmas de programação modular e programação orientada a objetos. A integração com todas as bibliotecas do LLVM foi implementada, com ajuda da comunidade *open source*, que se voluntariou no GoogleCode, mas ainda carece de testes.

Ao executar o LLVM-Pascal (com análise léxica, sintática completas e análise semântica parcial) ele alcança uma velocidade de 170 kloc/s, ao compilar as bibliotecas do Free Pascal/Lazarus em um ultrabook I7 de segunda geração, ou seja, seu tempo de execução é rápido como era estimado.

A versão já construída do LLVM-Pascal possui 12.233 loc (linhas de código), distribuídas da seguinte maneira:

- a) Analisador Léxico e Classe Token: 824 loc;
- b) Analisador Sintático: 239 loc;
- c) Analisador Semântico e Tabela de Símbolos: 238 loc;
- d) Gerador de Código: 97 loc;
- e) Gramática: 486 loc;
- f) Integração com LLVM: 10.213 loc;
- g) Programa Principal e rotinas utilitárias: 136 loc.

A construção do compilador foi complexa, exigindo mais tempo do que o foi previsto. Uma das maiores dificuldades foi construir e integrar as etapas do compilador, por causa da complexidade e interdisciplinaridade da tecnologia de compiladores. Outro dificultador foi o próprio LLVM. O LLVM é um *framework* extenso e para entendê-lo é requerido sólido conhecimento de programação em C++.

9.1 TRABALHOS FUTUROS

Considerando a complexidade do projeto serão necessárias tarefas futuras para a conclusão do compilador, conforme elencado abaixo:

- a) O frontend do compilador deverá ser *self-hosted*, ou seja, criado na linguagem Object Pascal e capaz de compilar a si mesmo;
- b) Concluir a integração com o LLVM (LLVM, 2012), como *backend* do compilador (gerador de código multiplataforma e otimizador);
- c) O LLVM-Pascal deverá ser capaz de compilar a RTL, FCL e LCL e ligá-las aos executáveis gerados;
- d) O compilador deverá ser maturado usando o *Test Suite* do Free Pascal (FREE PASCAL, 2012) e os conceitos usados na ferramenta Quest (QUEST-TESTER, 2012);
- e) Compilar o IDE Lazarus para Windows ou Linux como regra de passagem para alcançar a versão 1.0;
- f) Detalhar a estrutura e usos do LLVM-IR;
- g) Usar o Free Pascal e o Delphi como compiladores de referência para comparativos de:
 - Tempo de compilação;
 - Tamanho do executável gerado;
 - Performance de execução do executável gerado;
 - Expressividade das mensagens de erro e diagnóstico geradas durante a compilação;
 - Tamanho do compilador em linhas de código (só Free Pascal) e
 - Grau de compatibilidade com a linguagem Object Pascal.

REFERÊNCIAS BIBLIOGRÁFICAS

AHO, Alfred V.; MONICA S. LAM; RAVI SETHI; JEFFREY D. ULLMAN, **Compiladores: Princípios, técnicas e ferramentas**. São Paulo, 2a. edição, Pearson Addison-Wesley, 2008.

AHO, Alfred V.; RAVI SETHI; JEFFREY D. ULLMAN, **Compilers: Principles, Techniques and Tools**. Boston, Addison-Wesley, 1986.

AHO, Alfred V.; JEFFREY D. ULLMAN, **Compilers: Principles of Compiler Design**, Boston, Addison-Wesley, 1977.

AMAZON. **AMAZON.COM**. Disponível em: <http://www.amazon.com/Compilers-Principles-Techniques-Alfred-Aho/dp/0201100886/ref=sr_1_3?s=books&ie=UTF8&qid=1339165622&sr=1-3>. Acesso em 12 de maio de 2012.

CANNEYT, Michael Van. **Free Pascal : Reference guide**. Version 2.6.0. December 2011. Disponível em: <<http://www.freepascal.org/docs-html/ref/ref.html>>. Acesso em 12 de maio de 2012.

CRAIG, David T. **The Legacy of the Apple Lisa Personal Computer: An Outsider's View**. Disponível em: <<http://www.cs.oberlin.edu/~jwalker/lisa-legacy/>>. Acesso em 8 de junho de 2012.

EMBARCADERO. **Delphi XE2 Language Guide Index**. Disponível em: <http://docwiki.embarcadero.com/RADStudio/en/Delphi_Language_Guide_Index>. Acesso em 12 de maio de 2012.

FREE PASCAL. **Compiler Test Suite Results**. Disponível em: <http://wiki.freepascal.org/Compiler_test_suite_results>. Acesso em 8 de junho de 2012.

FREIRE, Paulo Guilherme Teixeira. **Micro Compilador de Expressão Aritmética**, IESB, Brasília, 2011.

FREITAS, Ricardo Luís de. **Compiladores**. Dezembro 2008. São Paulo: PUC-Campinas. Disponível em: <<http://code.google.com/p/compila/downloads/detail?name=Apostila%20de%20Compiladores%20EComp.pdf>>. Acesso em 20 de maio de 2012.

GOOGLECODE, **Support**. **User support for Google Project Hosting**. Disponível em: <<http://code.google.com/p/support/wiki/GettingStarted>>. Acesso em 8 de junho de 2012.

IESB. **IESB >> Graduação >> Ciência da Computação >> Matriz Curricular**. Disponível em: <<http://www.iesb.br/novosite/Home/graduacao/cienciaComputacao/matriz.php>>. Acesso em 8 de junho de 2012.

LAZARUS. **Lazarus**. Disponível em: <<http://www.lazarus.freepascal.org>>. Acesso em 12 de maio de 2012.

LLVM. **The LLVM Compiler Infrastructure**. Disponível em: <<http://llvm.org/>>. Acesso em 12 de maio de 2012.

LLVM-PASCAL. **LLVM-Pascal. Object Pascal Compiler for LLVM**. Disponível em: <<http://llvm-pascal.googlecode.com>>. Acesso em 12 de maio de 2012.

LOUDEN, Kenneth C. **Compiladores Princípios e Práticas**. São Paulo, Cengage Learning, 2004.

NEWTON, Isaac. **Letter to Robert Hooke**, February 1676.

OHLOH. **The Open Source Network**. Disponível em: <<http://www.ohloh.net/p/freepascal>>. Acesso em 8 de maio de 2012.

QUEST-TESTER, **Automatically test calling conventions of C compilers**. Disponível em: <<http://code.google.com/p/quest-tester/>>. Acesso em 8 de maio de 2012.

SOURCEFORGE, **SourceForge - Download, Develop and Publish Free Open Source Software**. Disponível em: <<http://sourceforge.net/>>. Acesso em 12 de maio de 2012.

TERRY, Patrick D. **Compilers and Compilers Generators**, 512 páginas, Coriolis Pub, March 1997.

TORRY. **Delphi Developers Library**. Disponível em: <<http://www.torry.net>>. Acesso em 8 de maio de 2012.

VALGRIND. **Valgrind Home**. Disponível em: <<http://valgrind.org/>>. Acesso em 8 de junho de 2012.

WIRTH, Nicklaus. **The Programming Language Pascal**. Pág. 35-63, Acta Informatica, 1971.

ANEXO A – GRAMÁTICA DO OBJECT PASCAL

```

! -----
! Delphi 7 Object Pascal Grammar
! -----
! (c) Rob F.M. van den Brink - the Netherlands, 2006 - R.F.M.vandenBrink@hccnet.nl
! Version V1.1, Aug 2006
!
! This grammar parses almost everything of the language, except for a few issues
! that are probably beyond the capabilities of Gold parser.
! Known limitations:
! (1) Cannot handle comment directives like {$ifdef Windows} ... {$endif}
!     When parts of the source code is uncommented in this way, the grammar will
!     still read it, and may fail.
! (2) The parser consumes all assembler statements, but is too tolerant in accepting
!     input. Groups of several <AsmItem> does not belong to a single <AsmInstruction>
!     because a 'newline' is regarded as whitespace while it should be a terminator here.
! (3) Lexemes like 'protected' , 'forward' can be both a keyword as well as an identifier
!     in Delphi (even in the same object declaration), and when these lexemes should mean
!     an identifier, the current grammar cannot handle it correctly.
!     For several of them a workaround was created, but a better solution should be
!     developed.
! (4) Strings with characters above #127 cannot be handled by the current Grammar.
!     This should be very simple, but if the grammar defines string characters
!     as the range {#32 .. #255}, Goldparser also adds {#376} and {#956}.
!     This looks like a bug in Gold Parser.
! (5) The inclusion of an adequate number of error productions (SynError) is still
!     to be done.
! (6) constructs that are considered (for the timebeing) as too weird, are not
!     supported; see below.
! -----
! This grammar supports also most of 'weird' constructs that Borland has added
! to Delphi. This refers to the inconsistent syntax for several directives
! like in <CallConvention> and <MethodDirective>. Sometimes these directives have
! to be separated by an ';' sometimes not and sometimes both is allowed.
! An example of a syntax that was considered as too weird to be covered by this
! grammar was found in library routine "IdSSLOpenSSLHeaders.pas" (comes with Delphi)
!   VAR
!   IdSslCtxSetVerifyDepth : procedure(ctx: PSSSL_CTX; depth: Integer); cdecl = nil;
!   IdSslCtxGetVerifyDepth : function (ctx: PSSSL_CTX):Integer; cdecl = nil;
! In a consistent syntax, cdecl should refer to some variable that is set to 'nil', but
! the ';' does not close the <TypeSpec> and the rest is still part of the syntax
!
! -----
! Version history
! V1.0 - June 2006, derived from scratch, using Delphi 7.0 help and code files
! V1.1 - Aug 2006, lots of refinements to cover almost everything of the language
! Consumes assembler code as well.
! -----

"Name"      = 'ObjectPascal'
"Version"   = '1.1, Aug 2006'
"Author"    = 'Rob F.M. van den Brink'
"About"     = 'Derived from scratch, from Delphi 7.0 help and experiments with code files'
"Start Symbol" = <Start>

! -----
"Case Sensitive" = False
"Virtual Terminals" = SynError CommentStart2 CommentEnd2

Comment Line      = '//'
Comment Start     = '{'
Comment End       = '}'

CommentStart1     = '('
CommentEnd1       = '*'

```

```

{Hex Digit}      = {Digit} + [abcdefABCDEF]
{Id Head}        = {Letter} + [_]
{Id Tail}        = {Id Head} + {Digit}
!{String Ch}     = {#32 .. #255} - [''] + {HT} !WHY DOES GOLFS ADD character #376 and #956 ??
{String Ch}      = {printable} - [''] + {HT}

DeclLiteral      = {digit}+
HexLiteral       = '$'{Hex Digit}+
FloatLiteral     = {Digit}+.{Digit}+
RealLiteral      = {Digit}+ ('.' {Digit}* | {Digit}*) ('E' '+' | 'E' '-' | 'E') {digit}+

StringLiteral    = ( '' {String Ch}* '' | '#' {digit}+ | '#' '$' {hex digit}+ | '''' )
+
!StringLiteral  = ( ( '' {String Ch}* '' )
!                  | ( '#' {digit}+ )
!                  | ( '#' '$' {hex digit}+ )
!                  | ( '''' )
!                  )+
id              = {Id Head}{Id Tail}*

```

```

<LCONST>        ::= DeclLiteral
<ICONST>        ::= DeclLiteral
                  | HexLiteral
<RCONST>        ::= FloatLiteral
                  | RealLiteral
<SCONST>        ::= StringLiteral
                  | '^' id      ! handles characters like ^H and ^V

```

!-----

```

!<UnitId>        ::= id
<IdList>         ::= <IdList> ',' <RefId>
                  | <RefId>
<LabelId>        ::= id
<TypeId>         ::= id
!                  | id '.' <RefId>
                  | NAME
<TypeName>       ::= <TypeId>
                  | id '.' <RefId>      !!accepts UnitId.Id as well
<TypeList>       ::= <TypeName>
                  | <TypeList> ',' <TypeName>
<RefId>          ::= id
                  | AT | ON | READ | WRITE | READLN | WRITELN | NAME | INDEX
                  | VIRTUAL | ABSOLUTE | MESSAGE | DEFAULT | OVERRIDE | ABSTRACT
                  | DISPID | REINTRODUCE

```

```

| REGISTER | PASCAL | CDECL | STDCALL | SAFECALL
| STRING   | WIDESTRING | ANSISTRING
| VARIANT  | OLEVARIANT
| READONLY | IMPLEMENTS | NODEFAULT | STORED
| OVERLOAD | LOCAL | VARARGS
| FORWARD
| CONTAINS | PACKAGE | REQUIRES | LIBRARY
| IMPORT   | EXPORT
| PLATFORM | DEPRECATED
| EXTERNAL

! PROTECTED | PUBLISHED | PRIVATE | PUBLIC !THIS FAILS IN FIELDS

<FieldDesignator> ::= <RefId>
| <FieldDesignator> '.' <RefId>

!* predefined identifiers are no part of a syntax
<RealTypeId> ::= REAL48
| REAL
| SINGLE
| DOUBLE
| EXTENDED
| CURRENCY
| COMP

<OrdTypeId> ::= SHORTINT
| SMALLINT
| INTEGER
| BYTE
| LONGINT
| INT64
| WORD
| BOOLEAN
| CHAR
| WIDECHAR
| LONGWORD
| PCHAR

*!

!-----
! M O D U L E S
!-----

<Start> ::= <Program> | <Unit> | <Package> | <Library>

<Program> ::= <ProgHeader> <OptUsesSection> <Block> '.'

<ProgHeader> ::= PROGRAM <RefId> <OptProgParamList> ';'

<OptProgParamList> ::= '(' <IdList> ')'
|

<Unit> ::= <UnitHeader> <InterfaceSection> <ImplementationSection>
<InitSection> '.'

<UnitHeader> ::= UNIT <RefId> <OptPortDirectives> ';'

<Package> ::= <PackageHeader> <OptRequiresClause> <OptContainsClause> END '.'

<PackageHeader> ::= PACKAGE <RefId> ';'

<OptRequiresClause> ::= REQUIRES <IdList> ';'
|

```

```

<OptContainsClause> ::= CONTAINS <IdList> ';'
                    |

<LibraryHeader>      ::= LIBRARY <RefId> ';'

<Library>            ::= <LibraryHeader> <OptUsesSection> <Block> '.'

<InterfaceSection>  ::= INTERFACE <OptUsesSection> <OptExportDeclList>

<OptUsesSection>    ::= <UsesSection>
                    |

<UsesClause>        ::= USES <IdList> ';'
                    | SynError

<UsesSection>       ::= <UsesClause>
                    | <UsesSection> <UsesClause>

<OptExportDeclList> ::= <ExportDeclList>
                    |

<ExportDeclList>    ::= <ExportDeclItem>
                    | <ExportDeclList> <ExportDeclItem>

<ExportDeclItem>    ::= <ConstSection>
                    | <TypeSection>
                    | <VarSection>
                    | <CallSection>
                    | <CallSection> FORWARD ';'
                    ! The forward directive has no effect in the interface section of a unit,
                    ! but is not forbidden here.

<CallSection>       ::= <ProcHeading>
                    | <FuncHeading>

<ImplementationSection> ::= IMPLEMENTATION <OptUsesSection> <OptDeclSection>
<OptExportBlock>

<InitSection>       ::= INITIALIZATION <StmtList> END
                    | INITIALIZATION <StmtList> FINALIZATION <StmtList> END
                    | <CompoundStmt>
                    | END

<Block>             ::= <OptDeclSection> <OptExportBlock> <CompoundStmt> <OptExportBlock>

<OptExportBlock>    ::= <ExportStmt>
                    | <OptExportBlock> <ExportStmt>
                    |

<ExportStmt>        ::= EXPORTS <ExportList> ';'

<ExportList>        ::= <ExportItem>
                    | <ExportList> ',' <ExportItem>

<ExportItem>        ::= id
                    | id NAME '' <ConstExpr> ''
                    | Id INDEX '' <ConstExpr> ''
                    | NAME '' <ConstExpr> ''
                    | INDEX '' <ConstExpr> ''

```



```

        | <CallType>          ';' <CallConventions> ';'
        | SynError           ';'

<Type> ::= <GenericType>
        | <CallType>

<TypeRef> ::= <TypeName>
            | <StringType>
            | <VariantType>

<GenericType> ::= <TypeName>
                | <StringType>
                | <VariantType>
                | <SubrangeType>
                | <EnumType>
                | <StructType>
                | <PointerType>
                | <ClassRefType>
                | <ClonedType>

<ClonedType> ::= TYPE <TypeRef>

<StringType> ::= STRING
                | ANSISTRING
                | WIDESTRING
                | STRING '[' <ConstExpr> ']'

<VariantType> ::= VARIANT
                | OLEVARIANT

<OrdinalType> ::= <SubrangeType> | <EnumType> | <TypeName>

<SubrangeType> ::= <ConstOrdExpr> '..' <ConstOrdExpr>
                | <ConstOrdExpr> SynError
                | '(' <RefId> ')' '..' <ConstOrdExpr>

<EnumType> ::= '(' <EnumList> ')'
            | '(' <RefId> ')'

<EnumList> ::= <EnumId>
            | <EnumList> ',' <EnumId>

<EnumId> ::= <RefId>
          | <RefId> '=' <ConstExpr>

<OptPacked> ::= PACKED
            |

<StructType> ::= <ArrayType>
                | <SetType>
                | <FileType>
                | <RecType>

<ArrayType> ::= <OptPacked> ARRAY '[' <OrdinalTypeList> ']' OF <Type>
            | <OptPacked> ARRAY OF CONST
            | <OptPacked> ARRAY OF <Type> !dynamic array, starting from 0
!      FCells: array of array of TIWGridCell;

<OrdinalTypeList> ::= <OrdinalType>
                    | <OrdinalTypeList> ',' <OrdinalType>

```

```

<RecType> ::= <OptPacked> RECORD <RecFieldList> END <OptPortDirectives>

<RecFieldList> ::=
    | <RecField1>
    | <RecField2>
    | <RecField1> ';' <RecFieldList>
    | <RecField2> ';' <RecFieldList>
    | <RecField2> ';' <CallConvention>
    | <RecField2> ';' <CallConvention> ';' <RecFieldList>
    | CASE <Selector> OF <RecVariantList>

<RecVariantList> ::=
    | <RecVariant>
    | <RecVariant> ';' <RecVariantList>

<RecField1> ::= <IdList> ':' <GenericType> <OptPortDirectives>

<RecField2> ::= <IdList> ':' <CallType>

<RecVariant> ::= <ConstExprList> ':' '(' <RecFieldList> ')'

<Selector> ::= <RefId> ':' <TypeName>
    | <TypeName>

<SetType> ::= <OptPacked> SET OF <OrdinalType>

<FileType> ::= <OptPacked> FILE OF <TypeRef>
    | FILE

<PointerType> ::= '^' <TypeRef>

<CallType> ::= PROCEDURE <OptFormalParams>
<OptCallConventions>
    | PROCEDURE <OptFormalParams> OF OBJECT
<OptCallConventions>
    | FUNCTION <OptFormalParams> ':' <ResultType>
<OptCallConventions>
    | FUNCTION <OptFormalParams> ':' <ResultType> OF OBJECT
<OptCallConventions>

!-----CLASSES AND OBJECTS-----

<RestrictedType> ::= <ObjectType>
    | <ClassType>
    | <InterfaceType>

<ObjectType> ::= <OptPacked> OBJECT <OptObjectHeritage> <ObjectMemberList> END

<ClassType> ::= CLASS <OptClassHeritage> <ClassMemberList> END
    | CLASS <OptClassHeritage>

<ClassRefType> ::= CLASS OF <TypeName>

<InterfaceType> ::= INTERFACE <OptClassHeritage> <OptClassGUID>
<OptClassMethodList> END
    | DISPINTERFACE <OptClassHeritage> <OptClassGUID> <OptClassMethodList>
END
    | INTERFACE
    | DISPINTERFACE

<OptObjectHeritage> ::= '(' <TypeName> ')'

```



```

|

<OptClassHeritage> ::= '(' <TypeList> ')'
|

<OptClassGUID> ::= '[' <ConstStrExpr> ']' ! <SCONST> globally unique identifier
|

<ObjectMemberList> ::=
| <ObjectMemberList> PUBLIC <OptFieldList> <OptObjectMethodList>
| <ObjectMemberList> PRIVATE <OptFieldList> <OptObjectMethodList>
| <ObjectMemberList> PROTECTED <OptFieldList> <OptObjectMethodList>

<ClassMemberList> ::=
| <ClassMemberList> PUBLIC <OptFieldList> <OptClassMethodList>
| <ClassMemberList> PRIVATE <OptFieldList> <OptClassMethodList>
| <ClassMemberList> PROTECTED <OptFieldList> <OptClassMethodList>
| <ClassMemberList> PUBLISHED <OptFieldList> <OptClassMethodList>

<OptFieldList> ::= <FieldList>
|

<OptObjectMethodList> ::= <ObjectMethodList>
|

<OptClassMethodList> ::= <ClassMethodList>
|

<FieldList> ::= <FieldSpec>
| <FieldList> <FieldSpec>

<ObjectMethodList> ::= <ObjectMethodSpec>
| <ObjectMethodList> <ObjectMethodSpec>

<ClassMethodList> ::= <ClassMethodSpec>
| <ClassMethodList> <ClassMethodSpec>

<FieldSpec> ::= <IdList> ':' <Type> <OptPortDirectives> ';'
| SynError ';'

<ObjectMethodSpec> ::= <MethodSpec> <OptMethodDirectives>
| <PropertySpec> <OptPropertyDirectives>
| SynError

<ClassMethodSpec> ::= <MethodSpec> <OptMethodDirectives>
| <ResolutionSpec> <OptMethodDirectives>
| CLASS <ProcSpec> <OptMethodDirectives>
| CLASS <FuncSpec> <OptMethodDirectives>
| <PropertySpec> <OptPropertyDirectives>
| SynError

<MethodSpec> ::= <ConstructorSpec>
| <DestructorSpec>
| <ProcSpec>
| <FuncSpec>

<ConstructorSpec> ::= CONSTRUCTOR <RefId> <OptFormalParms> ';'

<DestructorSpec> ::= DESTRUCTOR <RefId> <OptFormalParms> ';'

<ProcSpec> ::= PROCEDURE <RefId> <OptFormalParms> <OptCallConventions> ';'

<FuncSpec> ::= FUNCTION <RefId> <OptFormalParms> ':' <ResultType>
<OptCallConventions> ';'

<ResolutionSpec> ::= PROCEDURE <RefId> '.' <RefId> '=' <RefId> ';'

```

```

| FUNCTION    <RefId> '.' <RefId> '=' <RefId> ';'

<PropertySpec> ::= PROPERTY <PropertyDecl> <OptPropSpecifiers> ';'

<PropertyDecl> ::= <RefId> ':' <TypeRef>
| <RefId> '[' <IndexList> ']' ':' <TypeRef>
| <RefId>

<IndexList> ::= <IndexDecl>
| <IndexList> ';' <IndexDecl>

<IndexDecl> ::= <IdDecl>
| CONST <IdDecl>

<IdDecl> ::= <IdList> ':' <Type>

<OptPropSpecifiers> ::= <PropertySpecifier>
|

<PropertySpecifiers> ::= <PropertySpecifier>
| <PropertySpecifiers> <PropertySpecifier>

<PropertySpecifier> ::= INDEX <ConstExpr> !StorageSpecifier
| READ <FieldDesignator>
| WRITE <FieldDesignator>
| STORED <FieldDesignator>
! STORED <ConstExpr>
| DEFAULT <ConstExpr>
| NODEFAULT
| WRITEONLY
| READONLY
| DISPID <ConstExpr> !Only within InterfaceTypes
| <ImplementsSpecifier>

<ImplementsSpecifier> ::= IMPLEMENTS <TypeRef>
| <ImplementsSpecifier> ',' <TypeRef>
! The implements directive must be the last specifier in the property
! declaration and can list more than one interface, separated by commas

!-----VARS-----

<VarSection> ::= VAR <VarDeclList>
| THREADVAR <ThreadVarDeclList>

<VarDeclList> ::= <VarDecl>
| <VarDeclList> <VarDecl>

<VarDecl> ::= <IdList> ':' <Type> <OptAbsoluteClause> <OptPortDirectives> ';'
| <IdList> ':' <Type> '=' <TypedConstant> <OptPortDirectives> ';'
| <IdList> ':' <TypeSpec>
| SynError ';'

<ThreadVarDeclList> ::= <ThreadVarDecl>
| <ThreadVarDeclList> <ThreadVarDecl>

<ThreadVarDecl> ::= <IdList> ':' <TypeSpec>
| SynError ';'

<OptAbsoluteClause> ::= ABSOLUTE <RefId>
! ABSOLUTE <ConstExpr> !on windows only, not on linux
|

```

```

!-----
! E X P R E S S I O N S
!-----

<ConstExpr>          ::= <Expr>

<ConstOrdExpr>       ::= <AddExpr>

<ConstStrExpr>       ::= <AddExpr>

<Expr>               ::= <AddExpr>
                        | <AddExpr> <RelOp> <AddExpr>
                        | SynError

<AddExpr>            ::= <MulExpr>
                        | <AddExpr> <AddOp> <MulExpr>

<MulExpr>            ::= <Factor>
                        | <MulExpr> <MulOp> <Factor>

<Factor>             ::= NIL
                        | <ICONST>
                        | <RCONST>
                        | <SCONST>
                        | <Designator>
                        | <SetConstructor>
                        | '@' <Designator>
variable             | '@' '@' <Designator> !returns memory address of a procedural
                        | '(' <Expr> ')'
                        | '(' <Expr> ')' '^'
                        | '+' <Factor>
                        | '-' <Factor>
                        | NOT <Factor>
!---PortArray

<Designator>         ::= <FieldDesignator>
                        | <Designator> '.' <FieldDesignator>
                        | <Designator> '^'
                        | <Designator> '[' <ExprList> ']'
                        | <Designator> '(' <ExprList> ')' !FunctionCall or TypeCast
                        | <Designator> '(' ' ' ')' !FunctionCall
                        | <Designator> AS <TypeRef> !eg "with Source as TListItem do ..."
                        | '(' <Designator> ')'
                        | INHERITED <Designator>

<AsnOp>              ::= ':'
                        | '+=' | '-=' | '*=' | '/='

<RelOp>              ::= '=' | '>' | '<' | '<=' | '>=' | '<>'
                        | IN | IS | AS

<AddOp>              ::= '+' | '-'
                        | OR | XOR

<MulOp>              ::= '*' | '/'
                        | DIV | MOD | AND | SHL | SHR

<SetConstructor>     ::= '[' <SetElementList> ']'
                        | '[' ']'

<SetElementList>     ::= <SetElement>
                        | <SetElementList> ',' <SetElement>

```

```

<SetElement>      ::= <Expr>
                   | <Expr> '..' <Expr>

<ExprList>        ::= <Expr>
                   | <ExprList> ',' <Expr>

<FmtExpr>         ::= <Expr>
                   | <Expr> ':' <Expr>
                   | <Expr> ':' <Expr> ':' <Expr>

<FmtExprList>     ::= <FmtExpr>
                   | <FmtExprList> ',' <FmtExpr>

<ConstExprList>   ::= <ConstExpr>
                   | <ConstExprList> ',' <ConstExpr>

!-----
! S T A T E M E N T S
!-----

<StmtList>        ::= <Statement>
                   | <StmtList> ';' <Statement>

<Statement>       ::= <Label> <Statement>
                   | <AssignmentStmt>
                   | <CallStmt>
                   | <GotoStatement>
                   | <CompoundStmt>
                   | <IfStatement>
                   | <CaseStatement>
                   | <ForStatement>
                   | <WhileStatement>
                   | <RepeatStatement>
                   | <WithStatement>
                   | <TryFinallyStmt>
                   | <TryExceptStmt>
                   | <RaiseStmt>
                   | <AssemblerStmt>
                   | SynError
                   |

<Label>           ::= <LCONST> ':'
                   | <LabelId> ':'

<AssignmentStmt>  ::= <Designator> <AsnOp> <Expr>
                   | '@' <RefId> ':' <Factor>

!EXAMPLE of this '@' <RefId>, that calls the GetProcAddress
!function and points a var StrComp to the result.
! var StrComp: function(Str1, Str2: PChar): Integer;
!   ...
!   @StrComp := GetProcAddress(KernelHandle, 'lstrcmpi');

<CallStmt>        ::= <Designator> !procedure, function, method, typecast
                   | WRITE '(' <FmtExprList> ')'
                   | WRITELN '(' <FmtExprList> ')'
!                   | INHERITED <CallStmt>
                   | INHERITED

<GotoStatement>   ::= GOTO <LCONST>
                   | GOTO <RefId>

<CompoundStmt>    ::= BEGIN <StmtList> END

```

```

<IfStatement> ::= IF <Expr> THEN <Statement> ELSE <Statement>
               | IF <Expr> THEN <Statement>
               | IF SynError THEN <Statement>

<CaseStatement> ::= CASE <Expr> OF <CaseList> <Otherwise> END

<ForStatement> ::= FOR <RefId> '=' <Expr> <Dir> <Expr> DO <Statement>

<Dir> ::= TO | DOWNT0

<WhileStatement> ::= WHILE <Expr> DO <Statement>

<WithStatement> ::= WITH <DesignatorList> DO <Statement>

<DesignatorList> ::= <Designator>
                   | <DesignatorList> ',' <Designator>

<RepeatStatement> ::= REPEAT <StmtList> UNTIL <Expr>

<AssemblerStmt> ::= ASM <AsmLanguage> END

<Otherwise> ::= OTHERWISE <StmtList>
              | ELSE <StmtList>
              |

<CaseList> ::= <CaseSelector>
              | <CaseList> ';' <CaseSelector>
              | <CaseList> ';'

<CaseSelector> ::= <CaseLabels> ':' <Statement>

<CaseLabels> ::= <CaseLabel>
               | <CaseLabels> ',' <CaseLabel>

<CaseLabel> ::= <ConstExpr>
              | <ConstExpr> '..' <ConstExpr>

<RaiseStmt> ::= RAISE SynError ![object] [AT address]
              | RAISE <OptExceptInstance>
              | RAISE <OptExceptInstance> AT <Address>

<TryFinallyStmt> ::= TRY <StmtList> FINALLY <StmtList> END

<TryExceptStmt> ::= TRY <StmtList> EXCEPT <ExceptionBlock> <OptExceptionElse>
END

<ExceptionBlock> ::= <ExceptionStmt>
                   | <ExceptionBlock> ';' <ExceptionStmt>

<ExceptionStmt> ::= ON <Selector> DO <Statement>
!
                   | ELSE <Statement>
                   | <Statement>

<OptExceptionElse> ::= ELSE <StmtList>
                   |

<OptExceptInstance> ::= <Designator> !usually a method call (??)
                   |

<Address> ::= <Designator> !usually a function call, returning an address

```

```

<OptSemi>                ::= ';'
                           |

!-----
! R O U T I N E S
!-----

<ProcedureDeclSection> ::= <ProcedureDecl>
                           | <FunctionDecl>
                           | <MethodDecl>

<ProcedureDecl>          ::= <ProcHeading> <CallBody> <OptSemi>

<FunctionDecl>           ::= <FuncHeading> <CallBody> <OptSemi>

<MethodDecl>             ::= <MethHeading> <CallBody> <OptSemi>

<ProcHeading>            ::=          PROCEDURE          <RefId>          <OptFormalParams>
<OptCallSpecifiers> ';'  | <ProcHeading> <CallDirectives> <OptSemi>

<FuncHeading>            ::=          FUNCTION          <RefId> <OptFormalParams> ':' <ResultType>
<OptCallSpecifiers> ';'  |
                           | FUNCTION <RefId> ';'
                           | <FuncHeading> <CallDirectives> <OptSemi>
                           !
                           ! if the heading is 'incomplete' it was declared before
                           ! and must be followed by a <CallBody> TO BE IMPROVED

<MethHeading>            ::=  PROCEDURE          <RefId> '.' <RefId> <OptFormalParams>
<OptCallSpecifiers> ';'  |
                           | FUNCTION          <RefId> '.' <RefId> <OptFormalParams> ':'
<ResultType> <OptCallSpecifiers> ';'  |
                           | FUNCTION          <RefId> '.' <RefId> ';'
                           | CONSTRUCTOR      <RefId> '.' <RefId> <OptFormalParams>
<OptCallSpecifiers> ';'  |
                           | DESTRUCTOR      <RefId> '.' <RefId> <OptFormalParams>
<OptCallSpecifiers> ';'  |
                           | CLASS PROCEDURE <RefId> '.' <RefId> <OptFormalParams>
<OptCallSpecifiers> ';'  |
                           | CLASS FUNCTION <RefId> '.' <RefId> <OptFormalParams> ':'
<ResultType> <OptCallSpecifiers> ';'  |
                           | <MethHeading> <CallDirectives> ';'
                           !
                           ! the class methods operates on classes instead of objects

<ResultType>             ::= <TypeRef>

<OptFormalParams>        ::= '(' <FormalParamList> ')'
                           | '(' ')'
                           |

<FormalParamList>        ::= <FormalParam>
                           | <FormalParamList> ';' <FormalParam>

<FormalParam>            ::= <Parameter>
                           | CONST <Parameter>
                           | VAR <Parameter>
                           | OUT <Parameter>

<Parameter>              ::= <IdList>
                           | <IdList> ':' <ParmType>
                           | <IdList> ':' <TypeRef> '=' <ConstExpr>

```

```

<ParmType> ::= <TypeRef>
            | ARRAY OF <TypeRef>
            | ARRAY OF CONST
            | FILE

<CallBody> ::= <OptDeclSection> <CompoundStmt>
            | <OptDeclSection> <AssemblerStmt>
            | <ExternalDeclaration>
            | FORWARD

!-----DIRECTIVES-----
! <PortDirectives> are to produce warnings at compile time when source code is compiled
! in the {$HINTS ON} {$WARNINGS ON} state. It can be applied to declarations
!
! <PortDirectives are accepted:
!     WITHIN const and var declarations (not after),
!     WITHIN unit headers (not after)
!     AFTER function/procedure headings (not within) (ONE terminating ';' is optional)
! <PortDirectives are NOT accepted:
!     in/after type declarations
!-----

<OptPortDirectives> ::= <PortDirectives>
                    |

<PortDirectives> ::= <PortDirective>
                    | <PortDirectives> <PortDirective>

<PortDirective> ::= PLATFORM ! to warn that it is specific to a particular
operating environment (such as Windows or Linux)
                    | PLATFORM '=' <ConstExpr>
                    | DEPRECATED ! to warn it is obsolete or supported only for backward
compatibility
                    | LIBRARY ! to warn dependencies on a particular library or
component framework (such as CLX).

!-----

<OptMethodDirectives> ::= <MethodDirectives>
                    | <OptMethodDirectives> <PortDirective> ';'
                    |

<MethodDirectives> ::= <MethodDirective> ';'
                    | <MethodDirectives> <MethodDirective> ';'

<MethodDirective> ::= VIRTUAL
                    | VIRTUAL <ConstExpr>
                    | DYNAMIC !for classes only
                    | OVERRIDE
                    | ABSTRACT
                    | MESSAGE <ConstExpr>
                    | OVERLOAD
                    | REINTRODUCE
                    | DISPID <ConstExpr> !only within an <InterfaceType>
                    | <CallConvention>
                    !TODO NOT ALL THESE METHOD DIRECTIVES CAN BE COMBINED WITH EACH OTHER

!-----

<OptPropertyDirectives> ::= <PropertyDirective> ';'
                    | <OptPropertyDirectives> <PortDirective> ';'
                    |

<PropertyDirective> ::= DEFAULT

```

```

!-----
<ExternalDeclaration> ::= EXTERNAL
                        | EXTERNAL <ConstStrExpr>
                        | EXTERNAL <ConstStrExpr> NAME <ConstStrExpr>
!                        | EXTERNAL <SCONST>
!                        | EXTERNAL <SCONST> NAME <SCONST>

!-----

<CallDirectives>      ::= <CallDirective>
                        | <CallDirectives> <CallDirective>

<CallDirective>       ::= <CallConvention>
                        | <CallObsolete>
                        | <PortDirective>
                        | VARARGS      !works only with external routines and cdecl calling
convention.
                        | LOCAL        !prevents exporting in a library
                        | <SCONST>      !for PasPro only
                        | OVERLOAD

<OptCallSpecifiers>   ::= <CallSpecifier>
                        | <OptCallSpecifiers> <CallSpecifier>
                        |

<CallSpecifier>       ::= <CallConvention>
                        | <CallObsolete>

<CallConventions>     ::= <CallConvention>
                        | <CallConventions> <CallConvention>

<OptCallConventions>  ::= <CallConvention>
                        | <OptCallConventions> <CallConvention>
                        |

<CallConvention>      ::= REGISTER      ![ParmOrder]  [CleanUp]      [RegParms]
                        | PASCAL          !Left-to-right Routine    Yes
                        | CDECL           !Left-to-right Routine    No
                        | STDCALL         !Right-to-left Caller      No
                        | SAFECALL        !Right-to-left Routine     No

<CallObsolete>        ::= INLINE        !for backward compatibility only; has no effect on the
compiler.
                        | ASSEMBLER      !for backward compatibility only; has no effect on the
compiler.
                        | NEAR           !for 16-bits programming only, has no effect in 32 bit
applications
                        | FAR            !for 16-bits programming only, has no effect in 32 bit
applications
                        | EXPORT         !for 16-bits programming only, has no effect in 32 bit
applications

!-----
! A S S E M B L E R   I N S T R U C T I O N S
!-----
! The asm 'grammar' below is rather tolerant in accepting assembler code and needs
! further elaboration. The main problem is that end of lines are not detected, so
! the separators between the instructions cannot be found, so an <AsmItem> cannot
! be grouped in a meaningful way with another <AsmItem> into a true <AsmInstruction>
! A fundamental solution could be to start a new lexical "context" in this grammar, with
! that forces the lexical scanner to recognize an assembler specific list of keywords
! Unfortunately, this is not supported by the Gold parser

<AsmLanguage>         ::= <AsmInstruction>
                        | <AsmLanguage> <AsmInstruction>

```



```

<AsmInstruction>      ::= <AsmItem>
                        | <AsmInstruction> <AsmItem>
                        | <AsmInstruction> ',' <AsmItem>
                        | <AsmInstruction> ';'

<AsmItem>             ::= <AsmLabel>
                        | <AsmExpr>

<AsmLabel>            ::= <AsmLocal> ':'
                        | <AsmId>      ':'

<AsmExpr>             ::= <AsmFactor>
                        | '-' <AsmFactor>
                        | <AsmExpr> '+' <AsmFactor>
                        | <AsmExpr> '*' <AsmFactor>
                        | <AsmExpr> '-' <AsmFactor>
                        | <AsmExpr> '.' <AsmFactor>
                        | '[' <AsmExpr> ']'
                        | '(' <AsmExpr> ')'
                        | <AsmId> '(' <AsmExpr> ')'
                        | SynError

<AsmFactor>           ::= <AsmId>
                        | <AsmLocal>
                        | <ICONST>
                        | <RCONST>
                        | <SCONST>

<AsmId>               ::= <RefId>
                        | '&' <RefId>
                        | REPEAT | WHILE | IF
                        | AND | OR | XOR | SHR | SHL | DIV | NOT
                        | plus many other keywords as well that may serve as an identifier

<AsmLocal>            ::= '@' <LCONST>
                        | '@' <AsmId>
                        | '@' <AsmLocal>
                        | '@' END

```