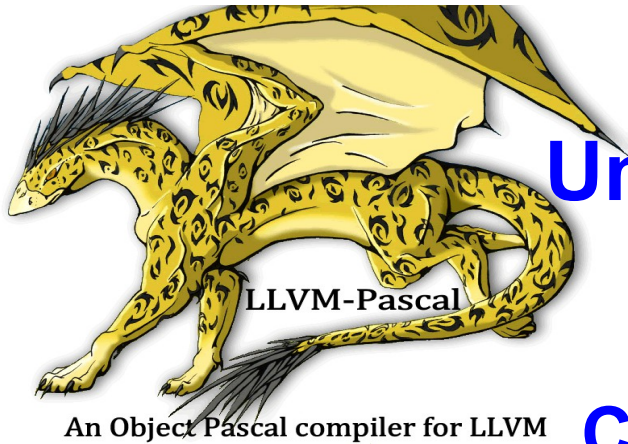


# Trabalho de Graduação



## LLVM-Pascal Um compilador Object Pascal usando LLVM

Ciência da Computação / IESB

Orientador: Prof. PhD Joel Guilherme da Silva Filho

[joel-iesb@joelguilherme.com](mailto:joel-iesb@joelguilherme.com)

Bárbara Alves Bezerra dos Anjos - [barbara.ab.anjos@gmail.com](mailto:barbara.ab.anjos@gmail.com)

Wanderlan Santos dos Anjos - [wanderlan.anjos@gmail.com](mailto:wanderlan.anjos@gmail.com)

Brasília, 08 de junho de 2012

# Tópicos da Apresentação

---

- O que é este trabalho?
  - Importância dos Compiladores.
  - O que é um compilador.
  - Qual a proposta do LLVM-Pascal
- O que é o LLVM?
  - LLVM – Low Level Virtual Machine.
- Como o LLVM se encaixa no nosso Compilador?
  - Análise x IR x Síntese
  - Fases do LLVM-Pascal.
  - Metodologia de desenvolvimento
  - Ambiente de desenvolvimento.
  - Desenvolvimento colaborativo.
  - Trabalhos futuros.

# O que é este trabalho?

- É o desenvolvimento do primeiro compilador para a linguagem Object Pascal usando um backend de mercado o LLVM (Low Level Virtual Machine) uma máquina virtual de baixo nível, da Apple, que é distribuído segundo os termos licenciamento do sistema de código aberto BSD (Berkeley Software Distribution).

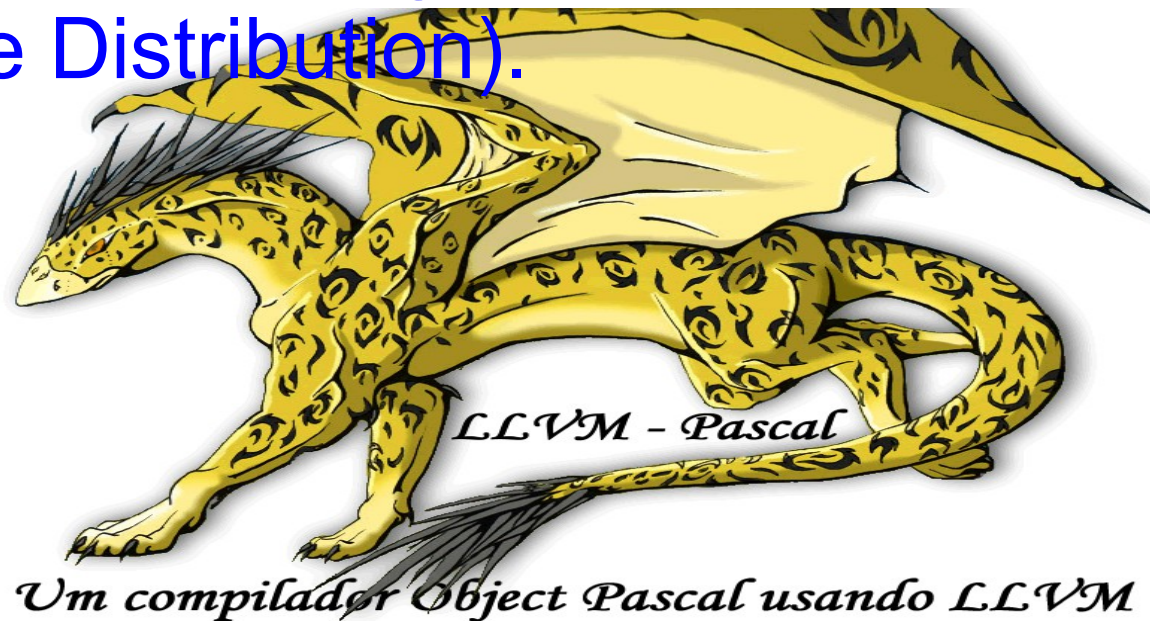


Figura 1 – Logo 1 do Compilador LLVM-Pascal 3

# O que é este trabalho?

- É criar um compilador simples, inteligível, facilmente gerenciável, num curto espaço de tempo, que possa ser usado em aplicações científicas e comerciais do mundo real e ainda como material de estudo para fins didáticos.

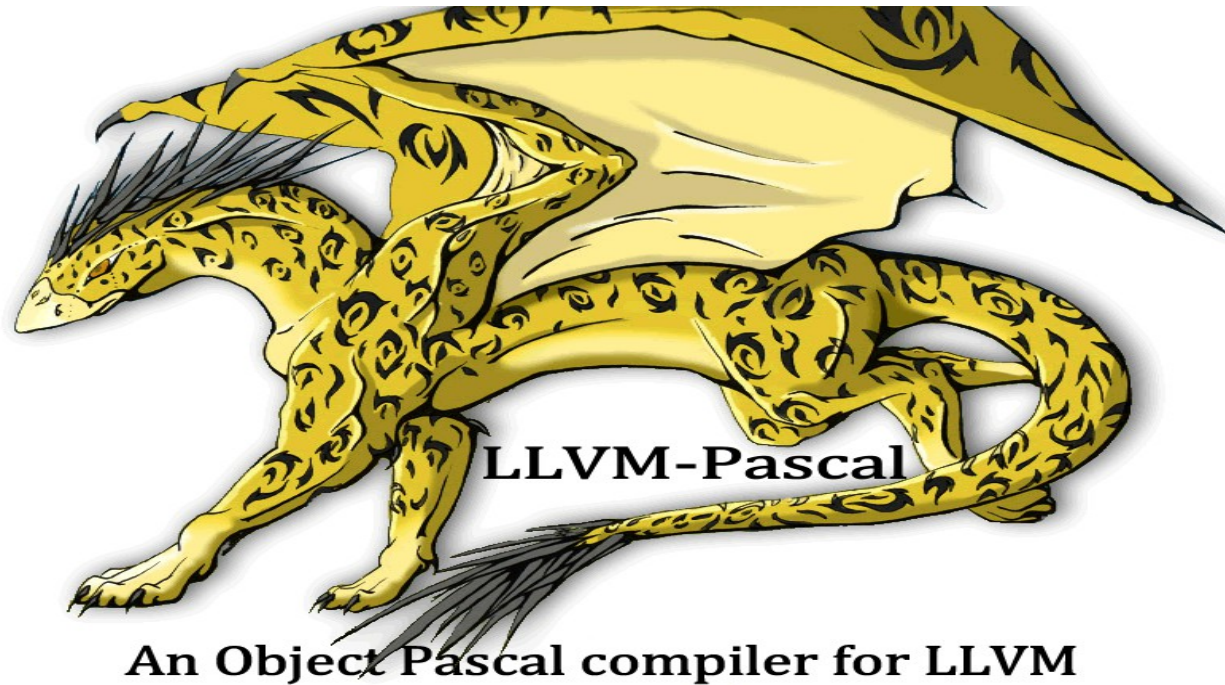


Figura 2 – Logo 2 do Compilador LLVM-Pascal 4

# Importância dos Compiladores

---

- As linguagens de programação estão presentes de forma direta nas máquinas que usamos hoje em dia.
- Com linguagens de programação podemos criar tecnologias e inovações para facilitar a vida das pessoas, simplesmente mudando o comportamento das máquinas usando linguagens de programação.

# Importância dos Compiladores

---

- O termo “linguagem de programação” deve ser entendido como uma notação textual, fácil de ser entendida por pessoas, que descreve as tarefas a serem executadas por um computador.
- Mas para que as máquinas entendam linguagens de programação é necessário um processo especial de tradução, chamado compilação, que traduz a notação textual, que é fácil de ser entendida por pessoas, para instruções em linguagem de máquina.

# O que é um compilador?

- Um compilador traduz uma linguagem de programação em linguagem de máquina.
- O compilador é o programa de computador capaz de realizar tais compilações.



Figura 3 – Visão geral de um compilador



# Complexidade de um compilador

- Escrever, entender e manter um compilador é uma tarefa árdua e desafiadora que, desde muito tempo, vem sendo comparada com à luta entre um cavaleiro medieval e um dragão cuspidor de fogo.

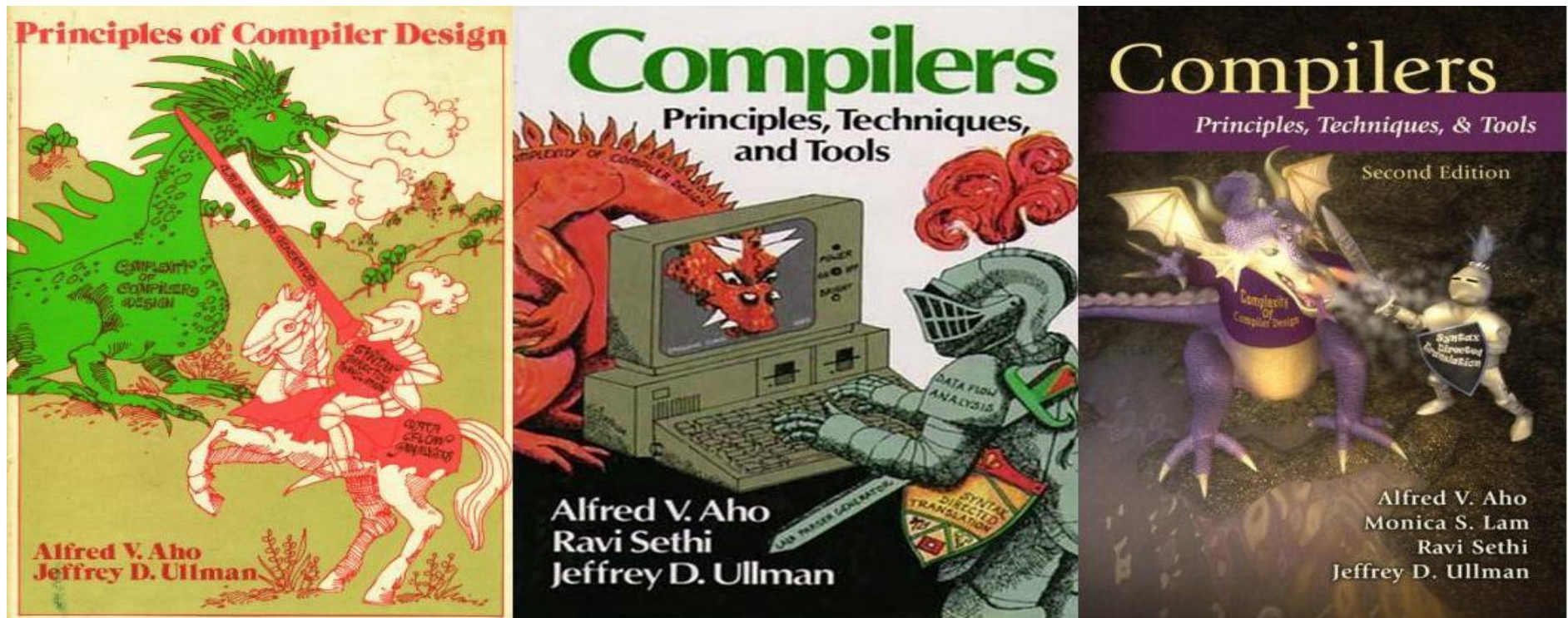


Figura 4 – Os três dragões da literatura de compiladores: green dragon book, red dragon book e purple dragon book.



# Qual a proposta do LLVM-Pascal?

---

- A criação de um compilador usando técnicas modernas e o princípio KISS (Keep it Short and Simple, ou seja minimalismo) para mantê-lo tão simples e gerenciável quanto possível.
- Um compilador similar possui 255.000 linhas de código. Nossa meta criar um muito menor.
- Gerar código mais performático.
- Que exiba mensagens de erro mais claras.
- Para conseguirmos esse objetivo usaremos o LLVM.

# O que é o LLVM?



Figura 5 – LLVM da Apple

# LLVM - Low Level Virtual Machine

- É um framework “estado-da-arte” 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. ([www.llvm.org](http://www.llvm.org))
- A partir de 2005 tornou-se um projeto estratégico da Apple.
- Apple o utiliza para compilar seus sistemas operacionais: Mac OS X e iOS.
- Com sua utilização mais da metade do esforço de construção de um compilador pode ser eliminada.

# Como o LLVM se encaixa no nosso Compilador?

- As duas principais fases do processo de compilação são chamadas de Análise e Síntese, que são respectivamente implementadas pelos módulos de front-end e back-end.
- O LLVM implementa todo o back-end do compilador.

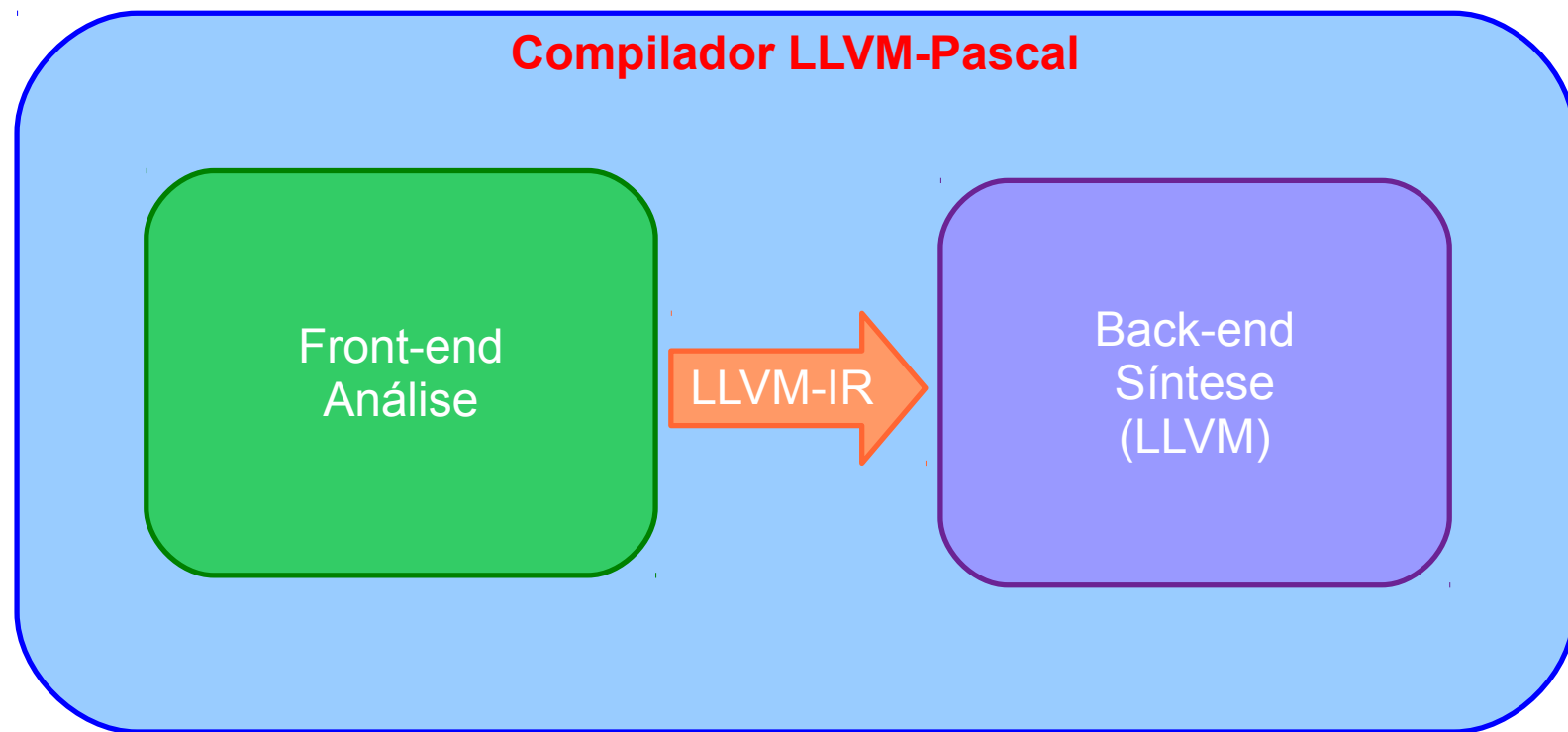


Figura 6 – Compilador LLVM - Pascal

# Análise x IR

- A análise é o processo desenvolvido no frontend: detecta os erros (léxicos, sintáticos e semânticos) no programa fonte e deverá emitir mensagens de erro para que o programador faça as correções necessárias.
- 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).



# Síntese x IR

---

- A partir da representação intermediária, que no nosso caso será a LLVM-IR, a parte de síntese tratará de gerar o código objeto, usando diversas subfases.
- O processo de síntese é desenvolvido no back-end e será inteiramente implementada pelo LLVM.

# Fases do LLVM-Pascal

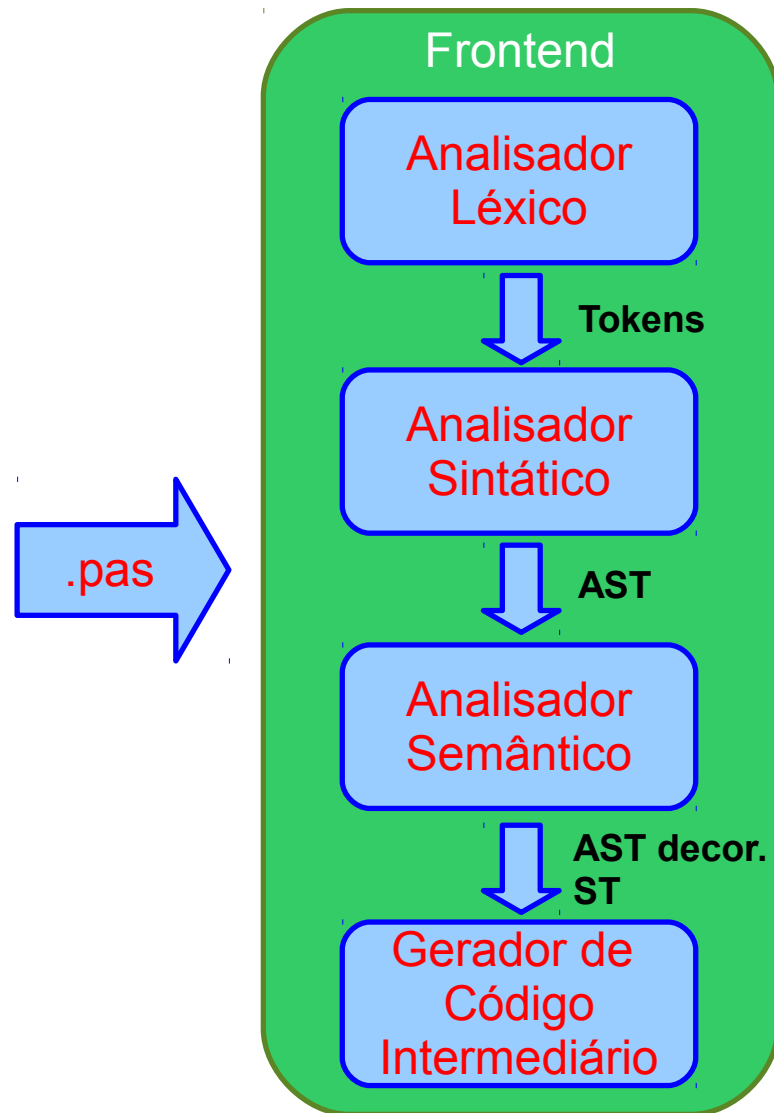


Figura 7 – Fases do compilador Frontend

# Analizador 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 8.
- 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.

# Analizador Léxico

- A sequência de tokens gerada pelo analisador léxico é utilizada como entrada do analisador sintático.

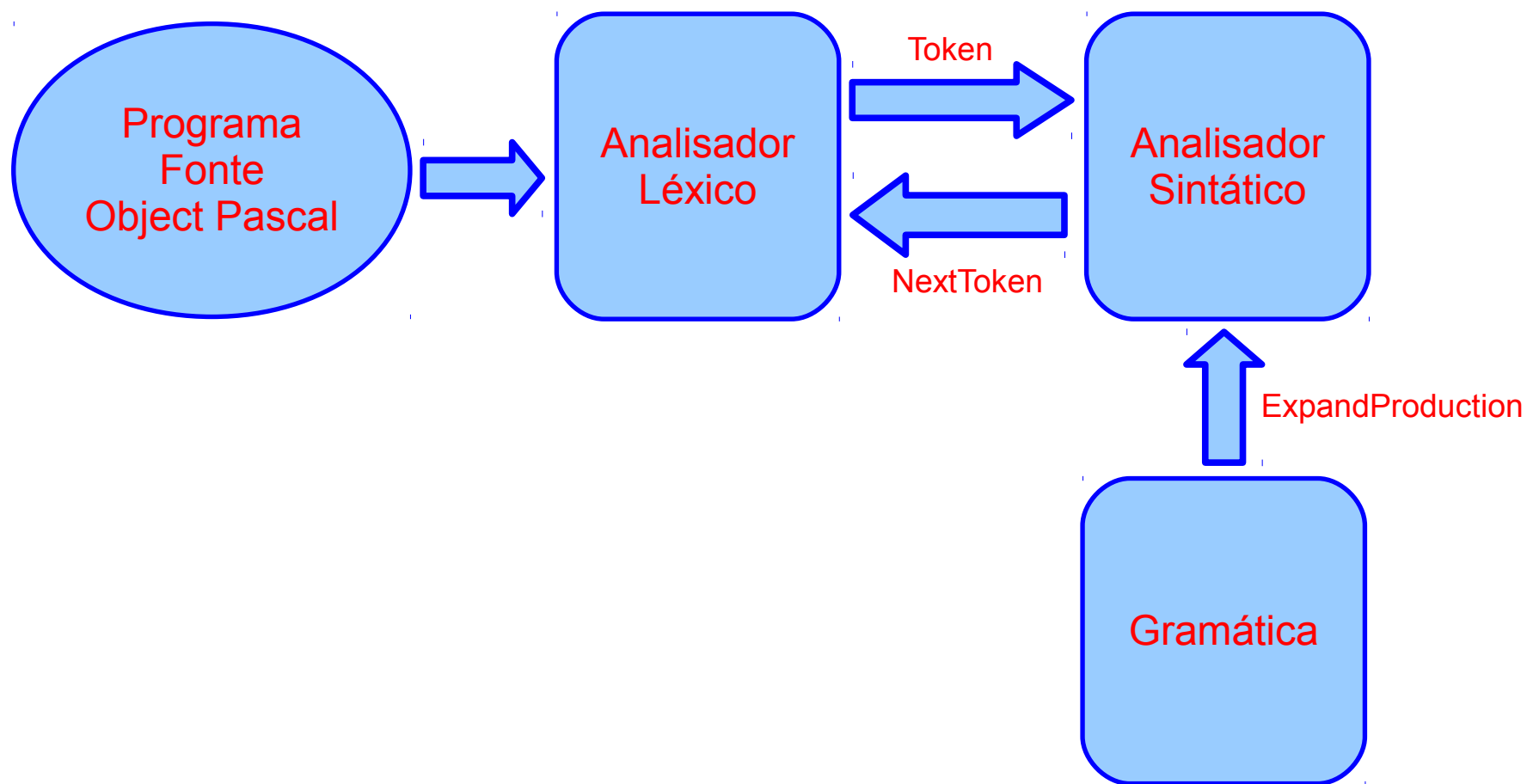


Figura 8 - Interface entre o analisador léxico e o sintático

# Analizador Léxico - 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 9.

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

Figura 9 - Classe TToken



# Analizador Léxico

---

- 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.

# Analizador Sintático

- É a segunda fase de um compilador e no caso do LLVM-Pascal é a mais importante.
- 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.

# Analizador Sintático

- Efeito colateral dessa verificação: Com base na AST o AS faz mais análises e posteriormente o gerador de código cria o código intermediário.

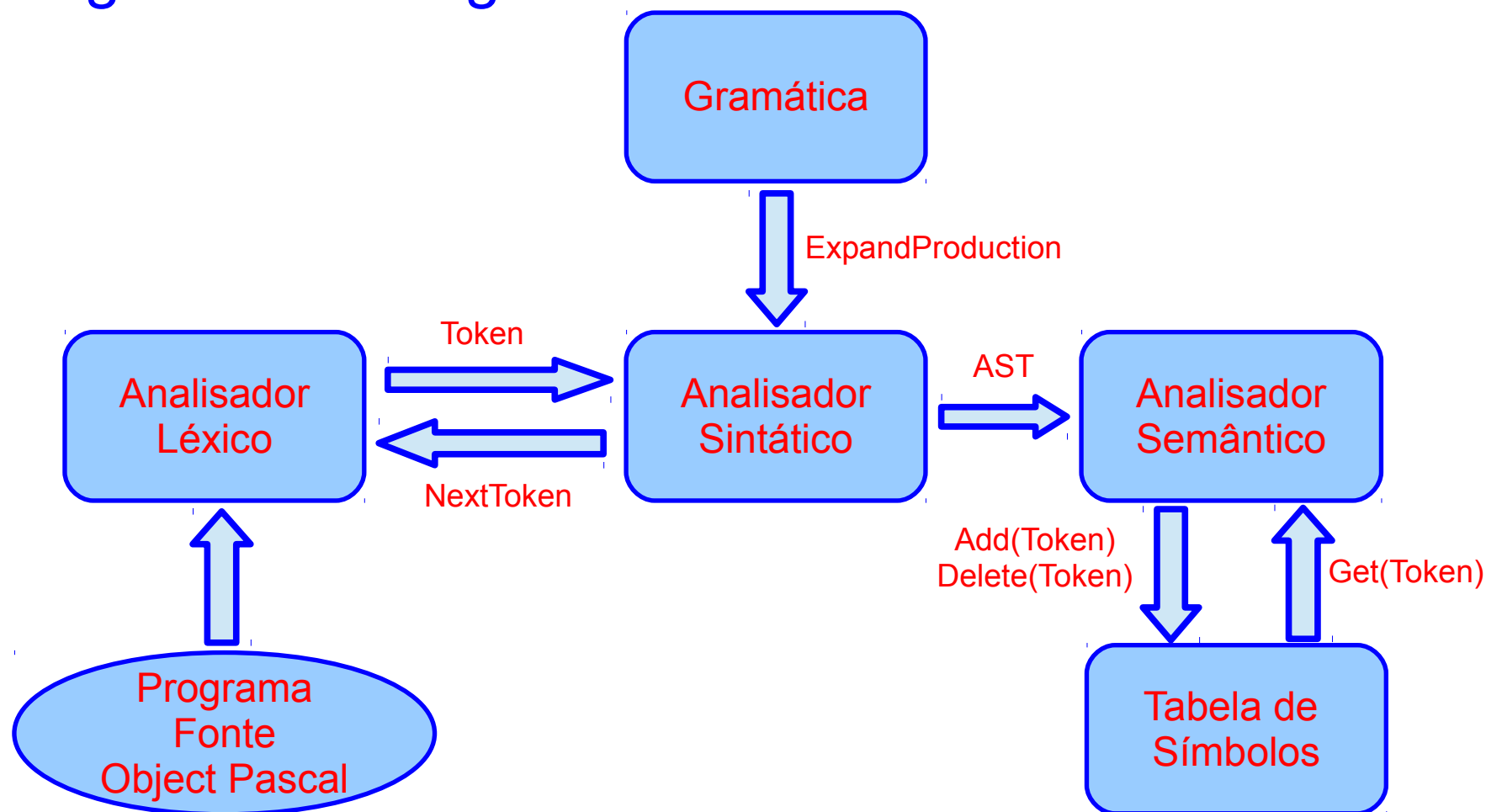


Figura 10 - Interação do analisador sintático

# Analizador Sintático x Complexidade

---

- Controle da complexidade do compilador
  - 120 produções = 120 métodos
  - LEX/YACC
  - 120 métodos → 5 métodos
  - Gramática em BNF transformada em SDT.
  - Gramática é parte do compilador
  - Máquina virtual de pilha.

# Analizador Sintático x 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.

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



# Analizador Sintático x Gramática

- 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 11.

<start> ::= PROGRAM <identificador>; <comandocomposto>.

<comandocomposto> ::= BEGIN <comando> END

<comando> ::= IF <expressão> THEN <comando>; | <comandocomposto>;

<expressão> ::= <identificador> [<operador> <identificador>]

Figura 11 – BNF do parágrafo anterior

# Analizador Sintático x 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.

# Analizador Sintático x 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.

# Analizador Sintático x Análise descendente

- A análise descendente ou top-down (LL, Left-Left) faz reconhecimento da linguagem, e produz uma derivação.

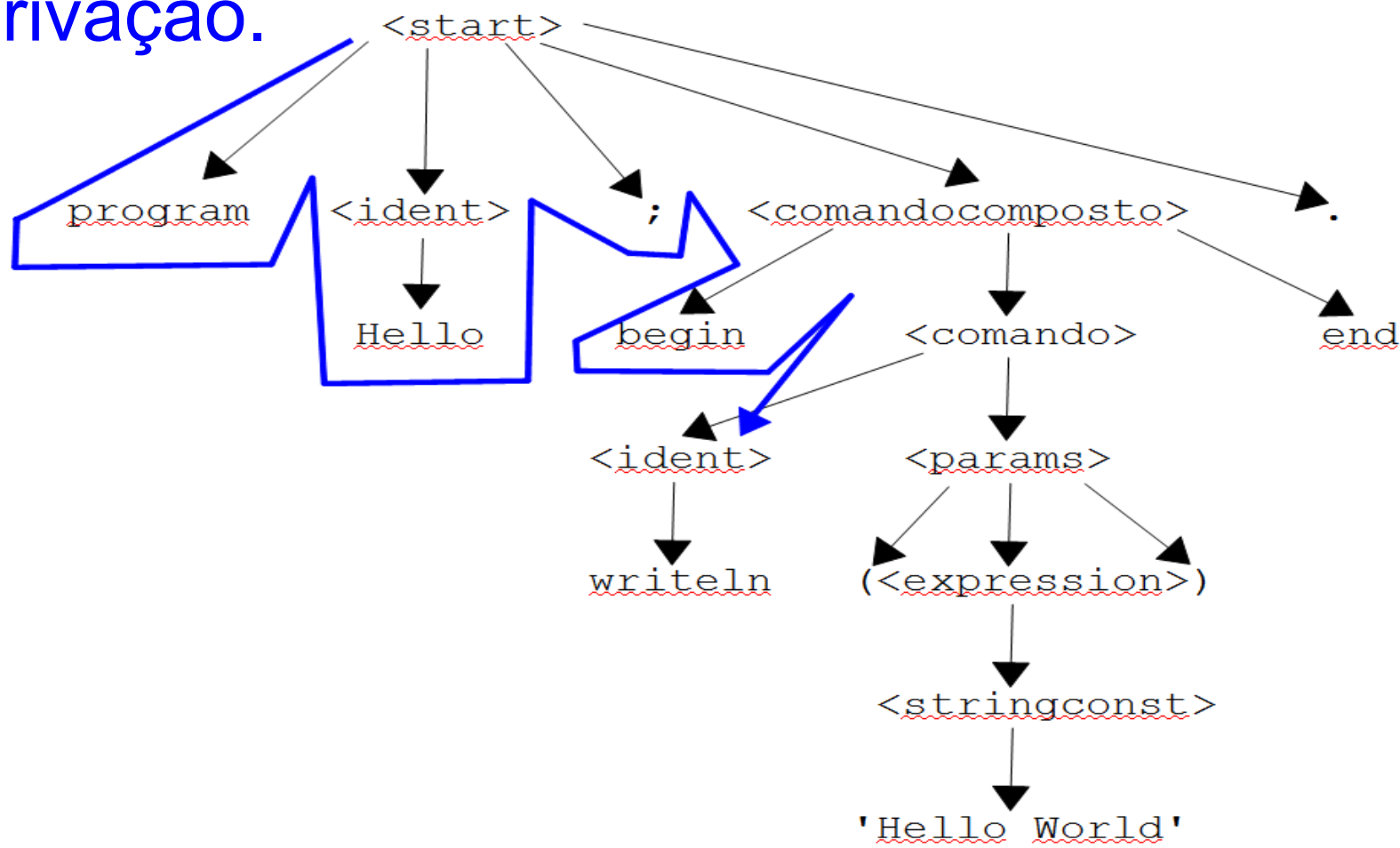


Figura 12 – Árvore de derivação - AST

# Analizador Sintático x Terminais

- 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.
- Definimos  $FIRST(A)$  como o conjunto de símbolos terminais que podem começar o símbolo não terminal.

$\langle \text{Start} \rangle ::= \langle \text{Program} \rangle | \langle \text{Unit} \rangle | \langle \text{Library} \rangle | \langle \text{Package} \rangle$

$\langle \text{Program} \rangle ::= \text{PROGRAM} \langle \text{Ident} \rangle ; \langle \text{DeclSection} \rangle \langle \text{CompoundStmt} \rangle .$

$\langle \text{Unit} \rangle ::= \text{UNIT} \langle \text{Ident} \rangle ; \langle \text{IntSection} \rangle \langle \text{ImplSection} \rangle \langle \text{InitSection} \rangle .$

$\langle \text{Library} \rangle ::= \text{LIBRARY} \langle \text{Ident} \rangle ; \langle \text{UsesClause} \rangle \langle \text{InitSection} \rangle .$

$\langle \text{Package} \rangle ::= \text{PACKAGE} \langle \text{Ident} \rangle ; \langle \text{Requires} \rangle \langle \text{Contains} \rangle \text{END} .$

Figura 13 – Gramática de exemplo para  $FIRST(\langle \text{Start} \rangle)$



# Analizador Sintático x SDT

- 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 13, convertida para SDT refatorada com terminais FIRST é mostrada na Figura 14.

```
// 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 14 – BNF transformada em SDT refatorada com terminais FIRST

# Analizador Sintático x Máquina 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:

```
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 15 – Algoritmo da máquina virtual de pilha

# Analizador Sintático x Máquina Virtual de Pilha

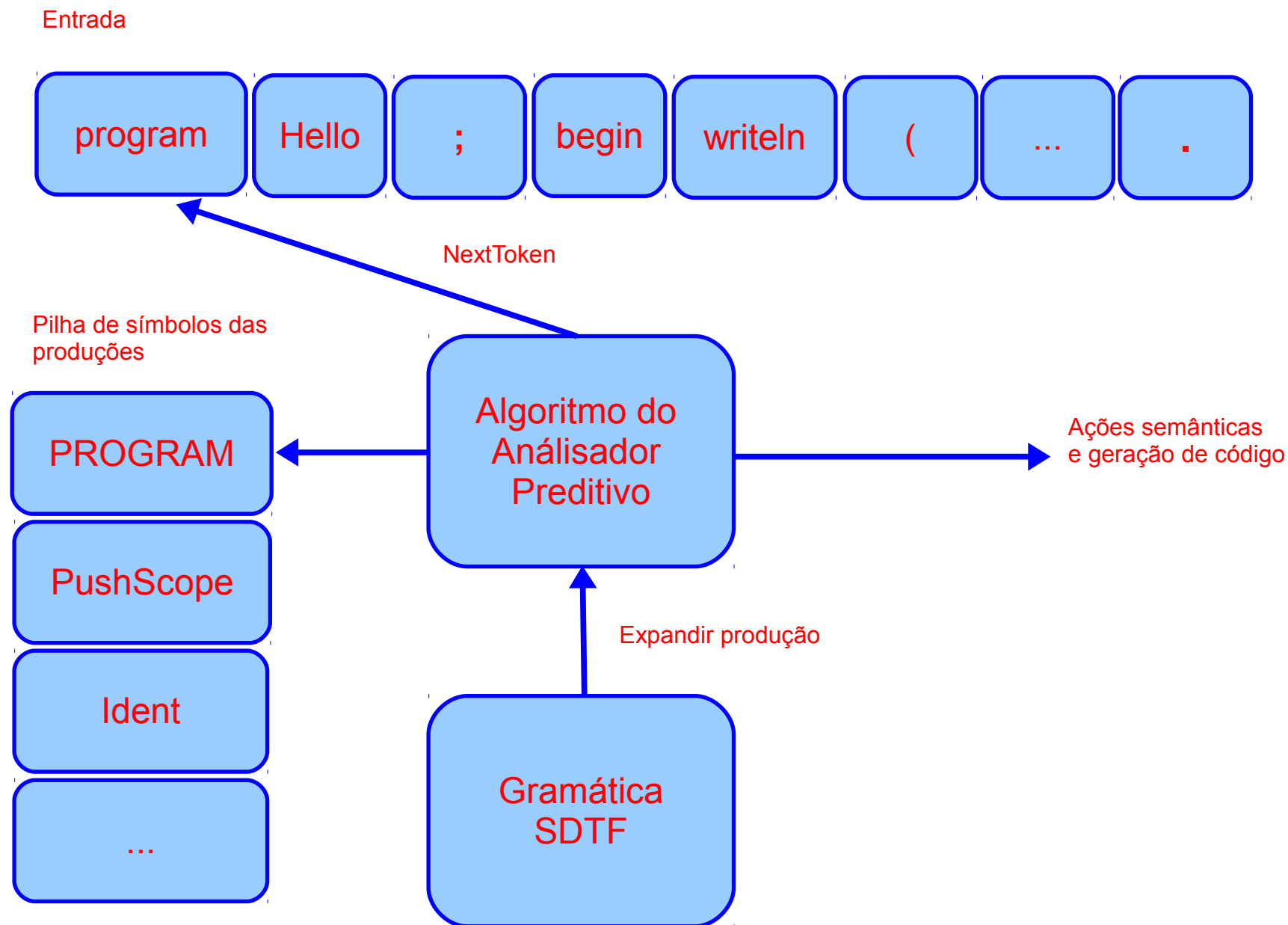


Figura 16 – Algoritmo da máquina virtual de pilha 31

# Analizador 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.

# Analizador Semântico

- 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);
  - d) Verificar se um identificador foi declarado no máximo uma vez em um escopo;
  - e) Verificar se um identificador não declarado está sendo usado;são foram respeitados;

# Analizador Semântico

---

- f) Verificar se um identificador não declarado está sendo usado;
- 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 foram respeitados;

# Analizador Semântico x Tabela de Símbolos

- 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`.

```
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.
```



# Analizador Semântico x Tabela de Símbolos

- Escopo1 as variáveis são inseridas normalmente na ST. Na entrada do Escopo2 o aninhamento garante que a cadeia de símbolos aplicáveis forme uma pilha.

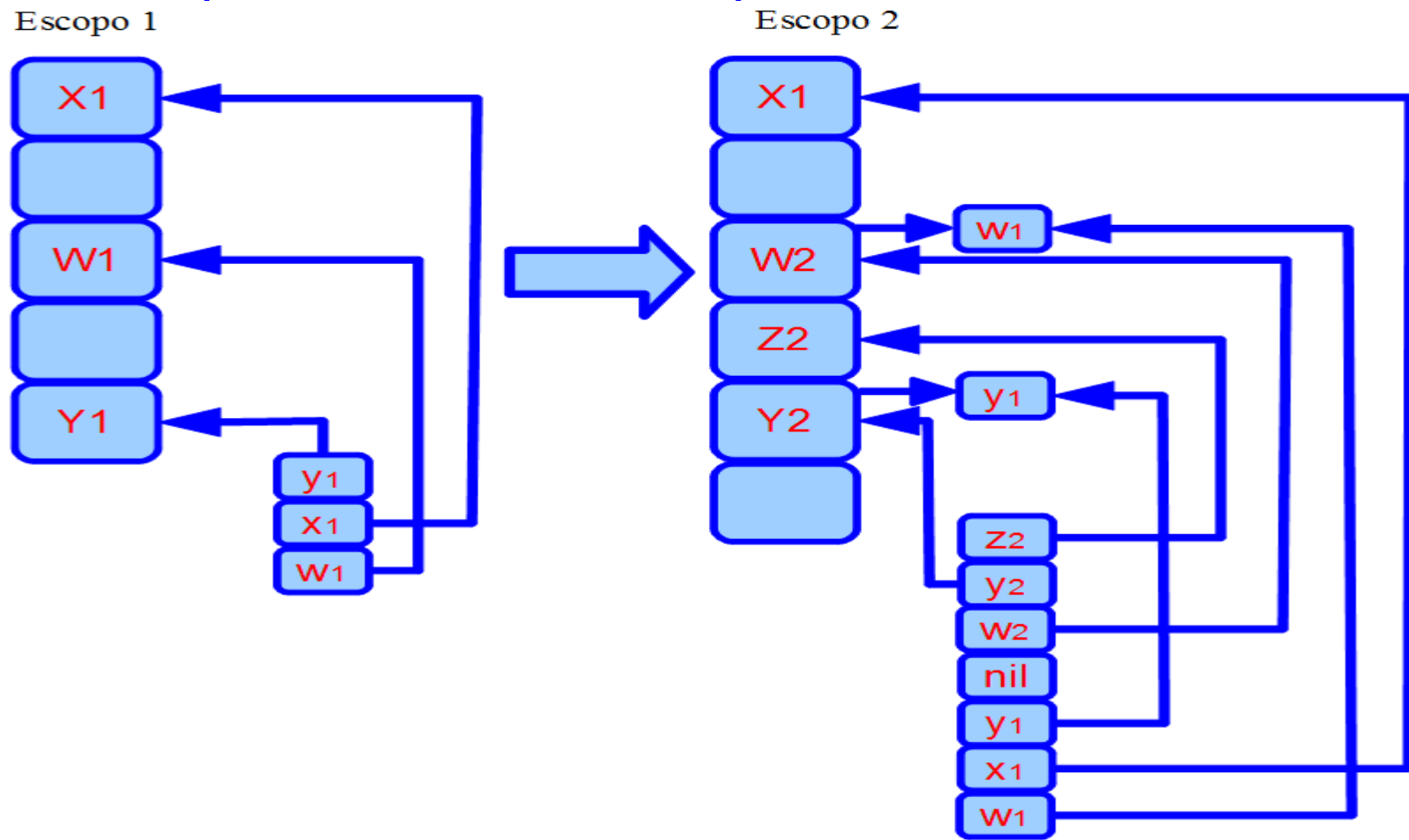


Figura 17 – Configuração da Tabela de Símbolos ao aninhar o escopo2 no escopo1

# 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).
- 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.

# Fases do LLVM-Pascal

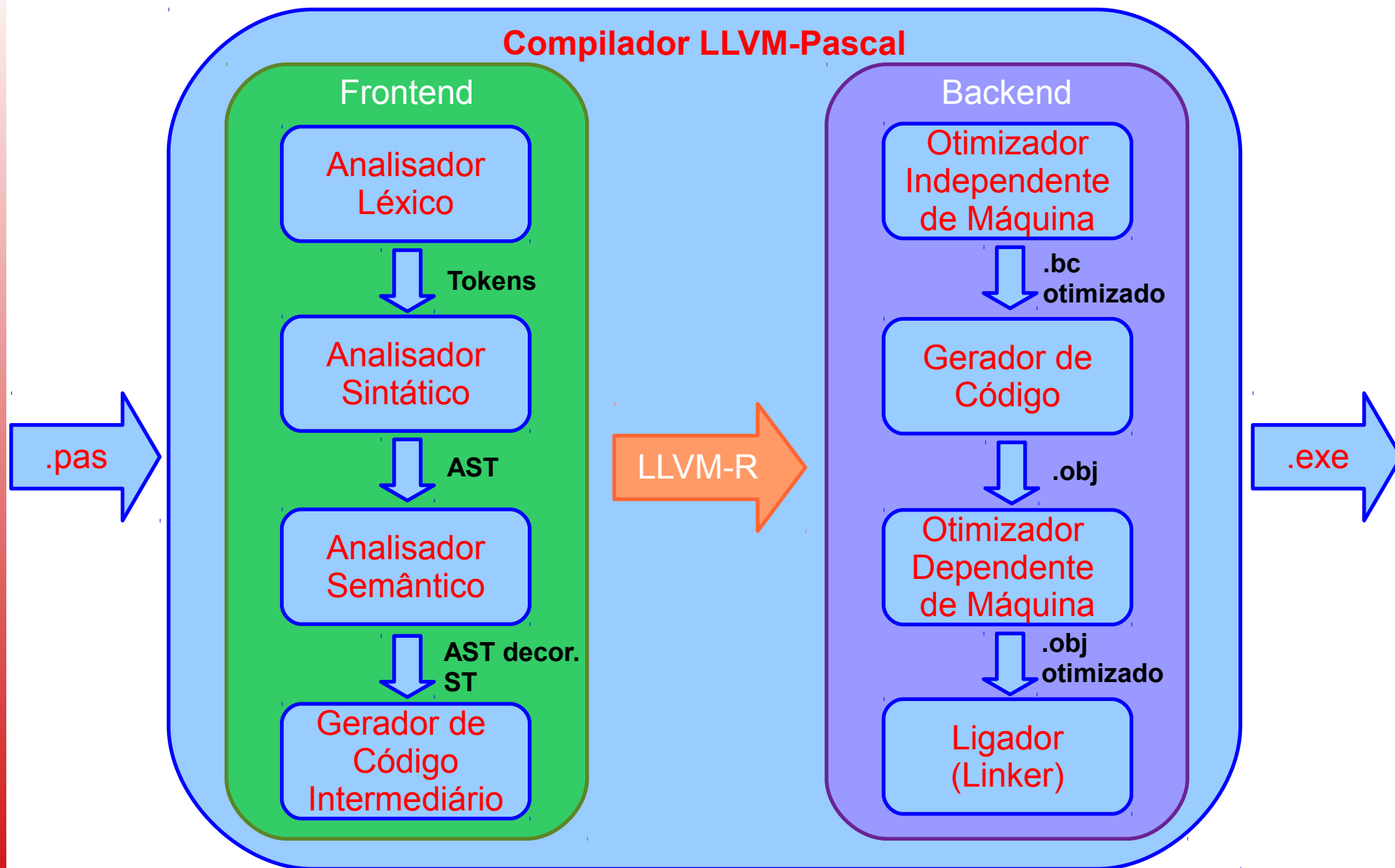


Figura 18 – Fases do compilador LLVM - Pascal

# Metodologia de desenvolvimento

---

- O frontend será desenvolvido em Object Pascal e será self-hosted, usaremos o Free Pascal como compilador bootstrap. Isso quer dizer, que a primeira versão do compilador será gerada pelo Free Pascal, que é o processo de bootstrapping.
- 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 diagrama T.

# Metodologia de desenvolvimento

---

- O frontend será desenvolvido em Object Pascal e será self-hosted, usaremos o Free Pascal como compilador bootstrap. Isso quer dizer, que a primeira versão do compilador será gerada pelo Free Pascal, que é o processo de bootstrapping.
- 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 diagrama T.

# Metodologia de desenvolvimento

- Um diagrama T 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.

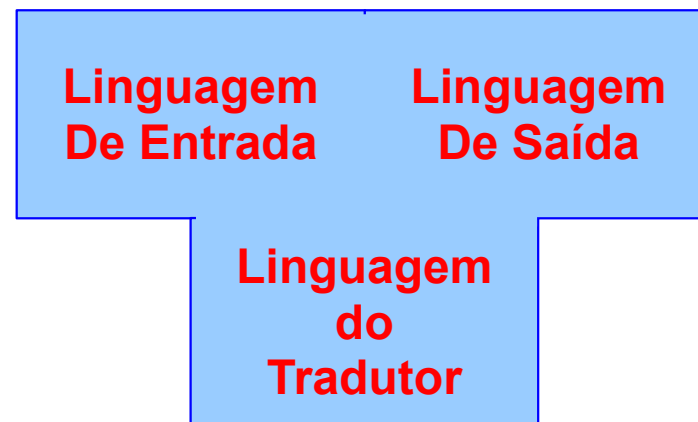


Figura 19 – Bloco de um diagrama T

# Metodologia de desenvolvimento

- O processo de bootstrapping do LLVM-Pascal, o T-diagram é composto por 3 tradutores.
- O primeiro é o 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 queremos gerar: um executável que traduz Object Pascal em bitcode.

# Metodologia de desenvolvimento

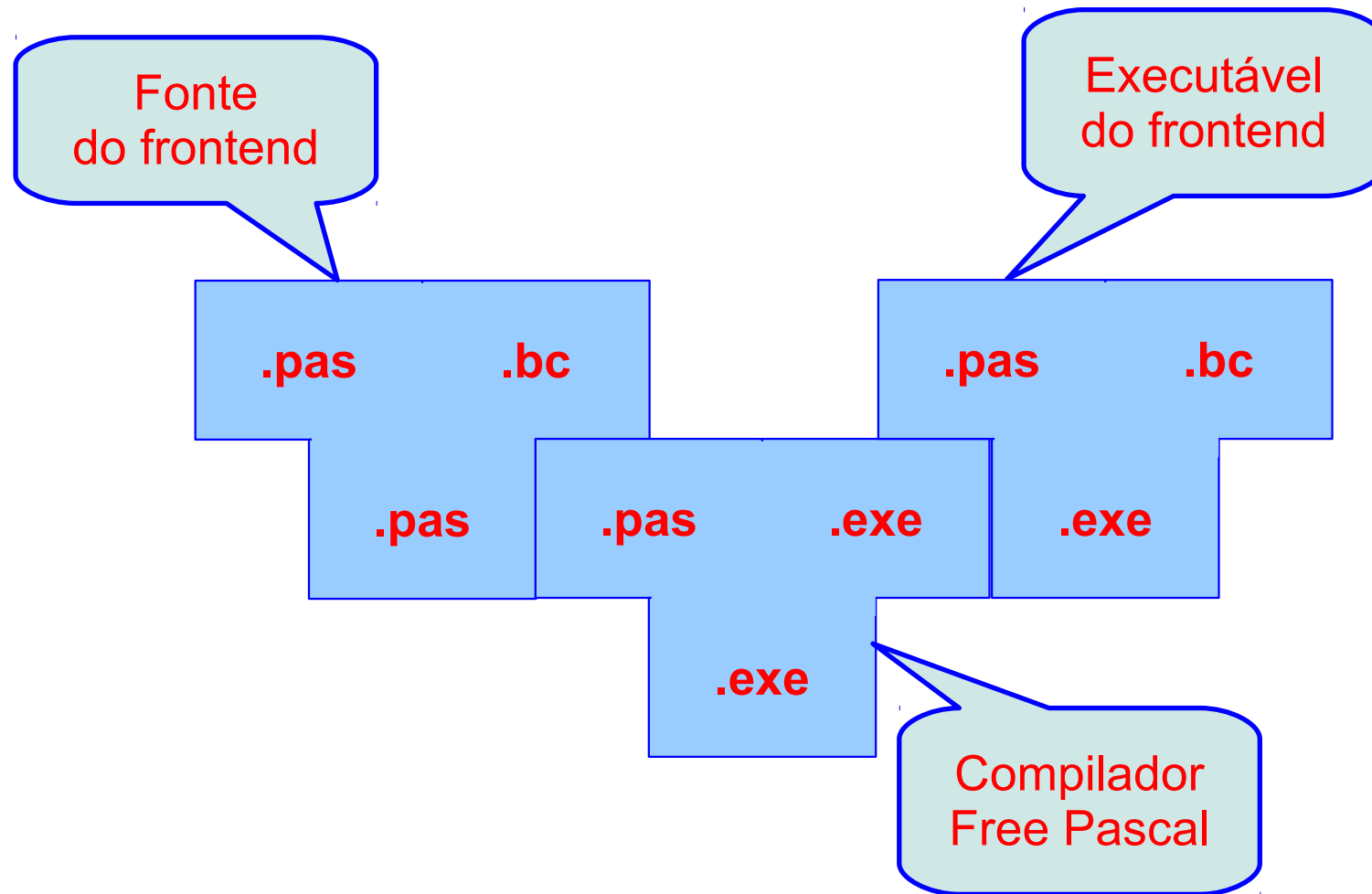


Figura 20 – Bootstrapping do LLVM-Pascal, usando Free Pascal



# Metodologia de desenvolvimento

- O próprio LLVM precisará ser bootstrapped pelo GCC conforme figura abaixo.

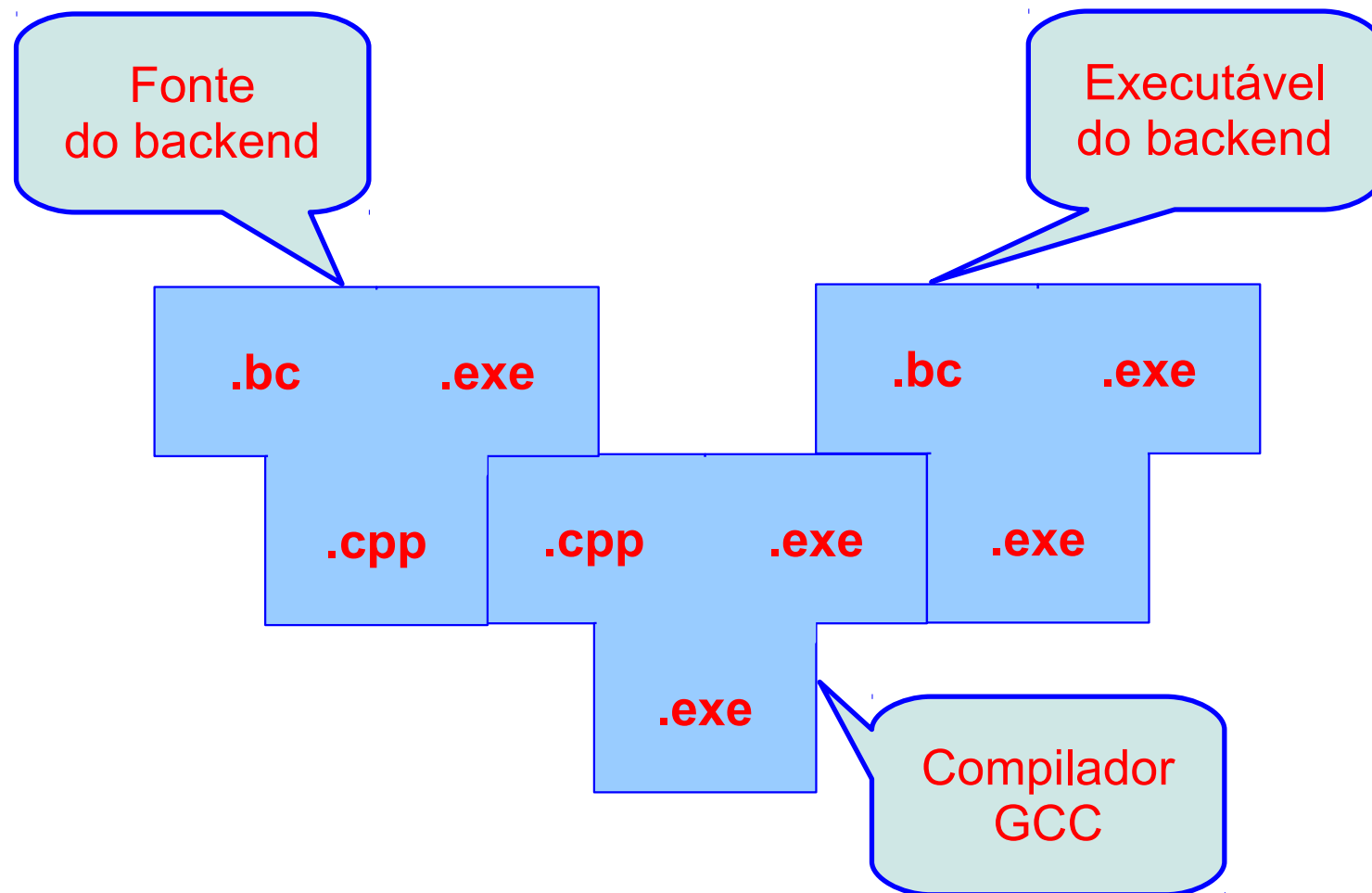


Figura 21 – Bootstrapping do backend LLVM, usando GCC

# Metodologia de desenvolvimento

- Após o bootstrapping do frontend e do backend teremos o compilador LLVM-Pascal como um executável.

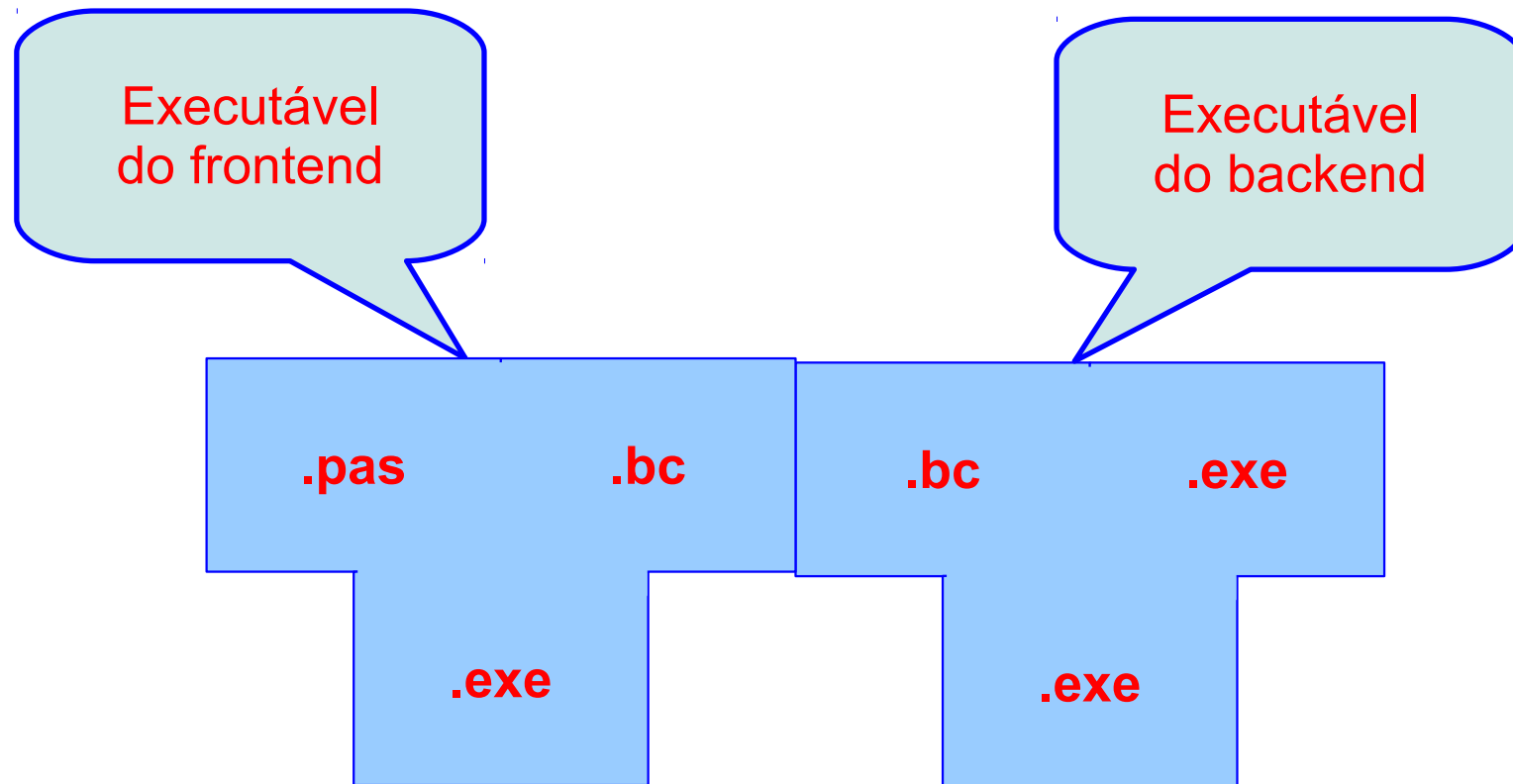


Figura 22 - O novo compilador LLVM-Pascal após o bootstrapping

# Metodologia de desenvolvimento

- 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.

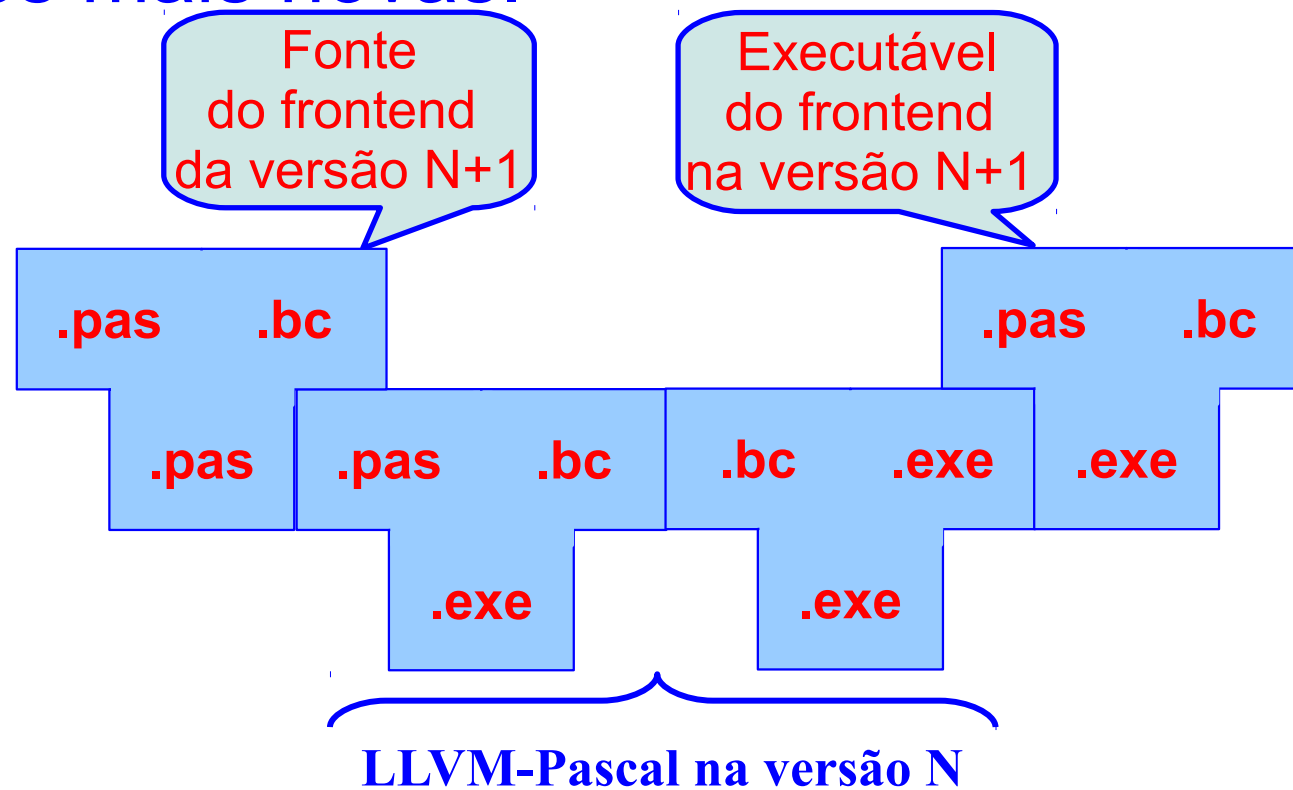


Figura 23 - Self-hosting do compilador migrando de uma versão N para N+1

# Ambiente de desenvolvimento

- Uma vez pronto, o usuário do LLVM-Pascal poderá usar o Lazarus, um poderoso IDE (Integrated Development Environment) licenciado como GPL, 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 realizável dentro do prazo de entrega deste TCC.

# Ambiente de desenvolvimento

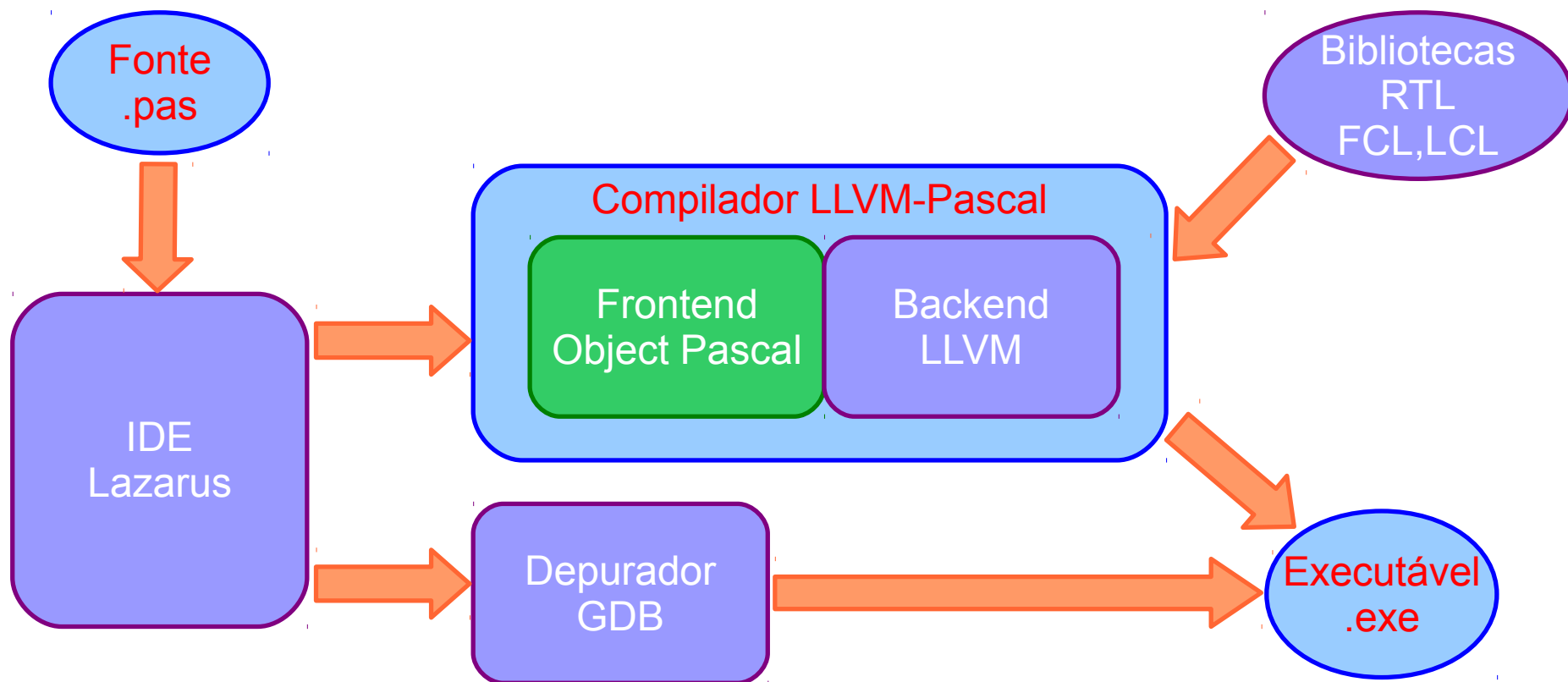


Figura 24 – Sistema de desenvolvimento com LLVM-Pascal

# Trabalhos Futuros

Outros compiladores, que não existem, que podem ser criados a partir do nosso projeto:

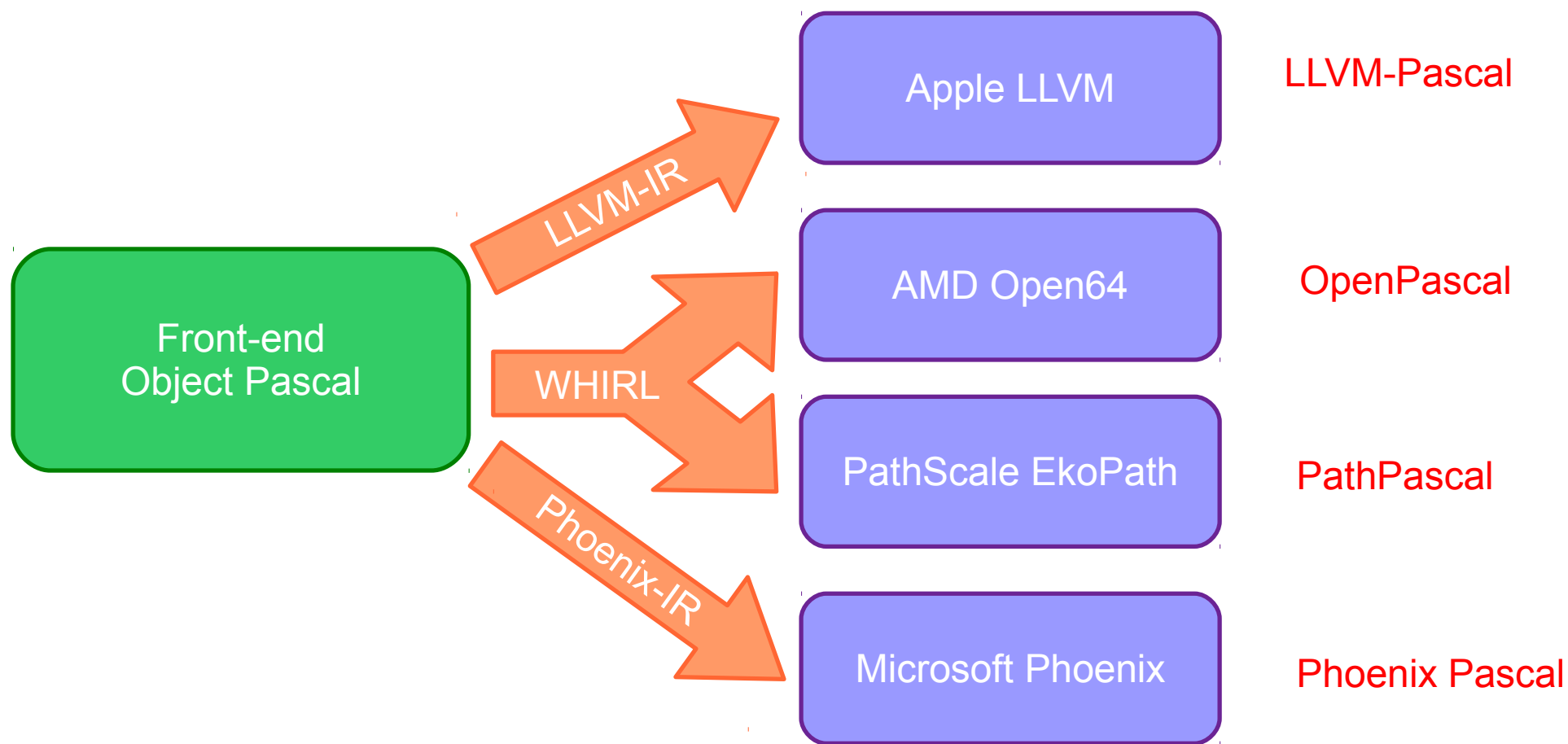
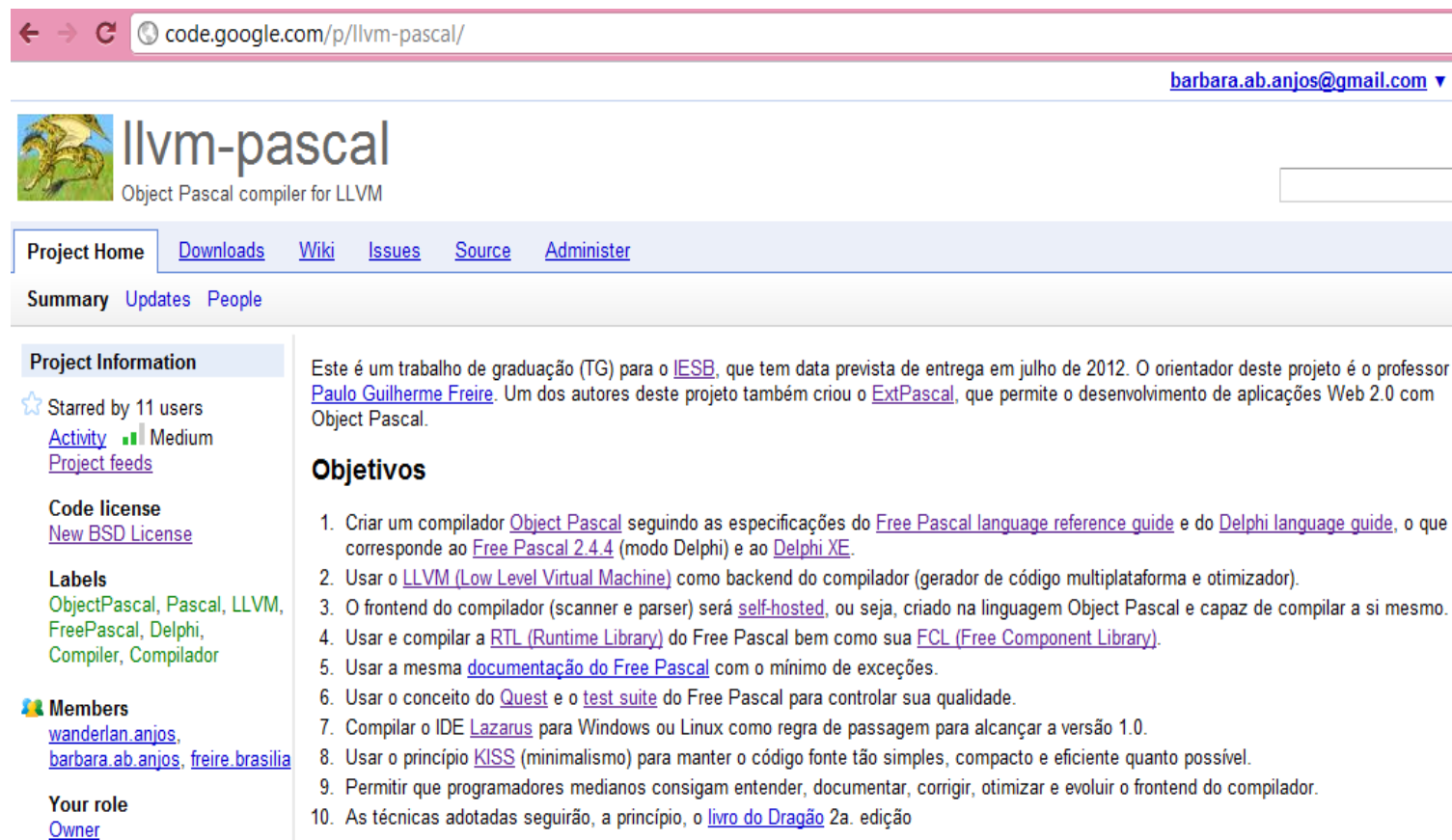


Figura 25 – Aplicações com Front-end Object Pascal

# Desenvolvimento colaborativo

- O desenvolvimento do LLVM-Pascal está on-line através do GoogleCode no site:  
<http://llvm-pascal.googlecode.com>



The screenshot shows the Google Code project page for LLVM-Pascal. The browser address bar displays `code.google.com/p/llvm-pascal/`. The project name is **llvm-pascal**, with the subtitle "Object Pascal compiler for LLVM". Navigation links include Project Home, Downloads, Wiki, Issues, Source, and Administer. The Summary tab is selected, showing project information, activity, and a list of members.

**Project Information**

- Starred by 11 users
- Activity: Medium
- Project feeds
- Code license: New BSD License
- Labels: ObjectPascal, Pascal, LLVM, FreePascal, Delphi, Compiler, Compilador
- Members: wanderlan.anjos, barbara.ab.anjos, freire.brasilia
- Your role: Owner

Este é um trabalho de graduação (TG) para o IESB, que tem data prevista de entrega em julho de 2012. O orientador deste projeto é o professor Paulo Guilherme Freire. Um dos autores deste projeto também criou o ExtPascal, que permite o desenvolvimento de aplicações Web 2.0 com Object Pascal.

**Objetivos**

1. Criar um compilador Object Pascal seguindo as especificações do Free Pascal language reference guide e do Delphi language guide, o que corresponde ao Free Pascal 2.4.4 (modo Delphi) e ao Delphi XE.
2. Usar o LLVM (Low Level Virtual Machine) como backend do compilador (gerador de código multiplataforma e otimizador).
3. O frontend do compilador (scanner e parser) será self-hosted, ou seja, criado na linguagem Object Pascal e capaz de compilar a si mesmo.
4. Usar e compilar a RTL (Runtime Library) do Free Pascal bem como sua FCL (Free Component Library).
5. Usar a mesma documentação do Free Pascal com o mínimo de exceções.
6. Usar o conceito do Quest e o test suite do Free Pascal para controlar sua qualidade.
7. Compilar o IDE Lazarus para Windows ou Linux como regra de passagem para alcançar a versão 1.0.
8. Usar o princípio KISS (minimalismo) para manter o código fonte tão simples, compacto e eficiente quanto possível.
9. Permitir que programadores medianos consigam entender, documentar, corrigir, otimizar e evoluir o frontend do compilador.
10. As técnicas adotadas seguirão, a princípio, o livro do Dragão 2a. edição

Figura 26 – Site do LLVM-Pascal

# Desenvolvimento colaborativo

---

- Com isso fica fácil e seguro a divulgação e o desenvolvimento do projeto num site central. Outros alunos ou colaboradores podem usar e participar do projeto bem como propor melhorias.
- O GoogleCode é similar ao SourceForge, porém muito mais fácil de usar, possui recursos de desenvolvimento colaborativo on-line, via Internet, para sistemas open source.



# Desenvolvimento colaborativo

---

- Vantagens de utilizar o GoogleCode:
  - Subversion: controle de versão dos fontes e documentação;
  - Download de arquivos do projeto: fontes, executáveis e documentação;
  - Páginas wiki para documentação on-line;
  - Bugtracking e controle de demandas e mudanças;
  - Links;
  - Clara apresentação da licença open-source;
  - Fórum de discussões.
  - Estatística de acesso: browser, Sistemas Operacional e país.

# Dificuldades

---

- Alta complexidade para construir e manter um compilador simples, inteligível, facilmente gerenciável.
- Interdisciplinaridade da tecnologia de compiladores.
- O LLVM é um frame-work extenso para estudo e para criação.
- Requer muita programação.

# 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.

# Conclusão

---

- Para tanto o compilador está claramente dividido em um frontend escrito em Object Pascal e em um backend, estado da arte, 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.
- Mais informações acesse o site do projeto em:

<http://llvm-pascal.googlecode.com>

# Dúvidas?

[llvm-pascal.googlecode.com](http://llvm-pascal.googlecode.com)



An Object Pascal compiler for LLVM

# Bibliografia

- 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-Princi>>. 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.

# Bibliografia

- EMBARCADERO. **Delphi XE2 Language Guide Index**. Disponível em: <  
[http://docwiki.embarcadero.com/RADStudio/en/Delphi\\_Language\\_](http://docwiki.embarcadero.com/RADStudio/en/Delphi_Language_)  
>. Acesso em 12 de maio de 2012.
- FREE PASCAL. **Compiler Test Suite Results**. Disponível em:  
<<http://wiki.freepascal.org/>>. Acesso em 8 de junho de 2012.
- FREIRE, Paulo Guilherme Teixeira. **Uma metodologia para desenvolvimento de compila-dores**: Compiladores Linguagem de programação construída para compiladores. Porto Alegre: Universidade Federal do Rio Grande do Sul, UFRGS, 1982. 191 páginas. Pós graduação em Ciência da Computação, Porto Alegre, Fevereiro de 1982.
- FREITAS, Ricardo Luís de. **Compiladores**. Dezembro 2008. São Paulo: PUC-Campinas. Disponível em: <  
<http://code.google.com/p/compila/downloads/detail?name=Apostila>  
>. Acesso em 20 de maio de 2012.



# Bibliografia

---

- **IESB. IESB >> Graduação >> Ciência da Computação >> Matriz Curricular.** Disponível em: <  
<http://www.iesb.br/novosite/Home/graduacao/cienciaComputac>  
>. Acesso em 8 de junho de 2012.
- **LLVM. The LLVM Compiler Infrastructure.** Disponível em: <  
<http://llvm.org/>>. 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