UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIA DA COMPUTAÇÃO – BACHARELADO

FERRAMENTA PARA EXECUÇÃO DE SCRIPTS OPL PARA OTIMIZAÇÃO COMBINATÓRIA EM GPU

ROGER WILLIAM WEBER

BLUMENAU
2013

2013/2-19

ROGER WILLIAM WEBER

FERRAMENTA PARA EXECUÇÃO DE SCRIPTS OPL PARA OTIMIZAÇÃO COMBINATÓRIA EM GPU

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Profa. Joyce Martins, Mestre - Orientadora

BLUMENAU
2013

2013/2-19

FERRAMENTA PARA EXECUÇÃO DE SCRIPTS OPL PARA OTIMIZAÇÃO COMBINATÓRIA EM GPU

Por

ROGER WILLIAM WEBER

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Joyce Martins, Mestre – Orientadora, FURB
Membro:	Prof. Dalton Solano dos Reis, Mestre – FURB
Membro:	Prof. Aurélio Faustino Hoppe, Mestre – FURB

Blumenau, 28 de janeiro de 2014

Dedico este trabalho à minha família e a todos os amigos, especialmente aqueles que me ajudaram diretamente na realização deste.

AGRADECIMENTOS

À minha família, em especial à minha mãe, pelo constante incentivo nos estudos e na conclusão deste trabalho.

À minha orientadora, Joyce Martins, por ter acreditado na conclusão deste trabalho.

Aos meus amigos, pelos incentivos e cobranças.

RESUMO

Este trabalho apresenta a especificação e o desenvolvimento de uma ferramenta de compilação de códigos OPL para, com programação linear, resolver problemas de otimização combinatória, gerando programas equivalentes em linguagem C++ para execução em placa de vídeo NVIDIA CUDA. Foi implementada uma versão do algoritmo Simplex em C++. É detalhada a linguagem OPL, assim como a ferramenta de desenvolvimento IBM ILOG CPLEX. Os resultados obtidos mostram que a ferramenta é capaz de processar um subconjunto da linguagem OPL gerando código equivalente. Este por sua vez produz resultados satisfatórios comparados à ferramenta IBM ILOG CPLEX.

Palavras-chave: CUDA. OPL. C++. Compilador.

ABSTRACT

This paper presents the specification and development of a tool to compile OPL codes, with linear programming, solving problems of combinatorial programming, generating equivalent programs in C++ to run on the NVIDIA CUDA. Was implemented a version of the Simplex algorithm in C++. It detailed the OPL language as well as the development tool IBM ILOG CPLEX. The results show that the tool is able to process a subset of the OPL language generating equivalent code. This in turn produces satisfactory results compared to the IBM ILOG CPLEX tool.

Key-words: CUDA. OPL. C++. Compiler.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SIGLAS

AMPL – *A Mathematical Programming Language*

CPU – *Central Processing Unit*

CUDA – *Compute Unified Device Architecture*

GAMS – *General Algebraic Modeling System*

GPGPU – *General Purpose Graphics Processing Unit*

GPU – *Graphics Processing Unit*

IDE – *Integrated Development Environment*

NVCC – *NVIDIA's CUDA Compiler*

OPL – *Optimization Programming Language*

SIMD – *Single Instruction Multiple Data*

SUMÁRIO

1 INTRODUÇÃO

Otimização combinatória é um ramo da ciência da computação e da matemática aplicada que estuda problemas de otimização em conjuntos finitos. Em um problema de otimização combinatória tem-se uma função objetivo e um conjunto de restrições, ambos relacionados às variáveis de decisão. Os valores possíveis das variáveis de decisão são delimitados pelas restrições impostas sobre estas variáveis, formando um conjunto discreto (finito ou não) de soluções factíveis a um problema. Segundo Nazareth (2004, p. 1), otimização é a arte, a ciência e a matemática de encontrar o melhor membro de um conjunto finito ou infinito de possíveis escolhas e com base em algum objetivo mensurar o mérito de cada escolha no conjunto.

Vários problemas, de acordo com IBM Software (2011a), podem ser resolvidos usando otimização combinatória, podendo citar otimização de transporte, logística, produção, gerenciamento de riscos, alocação de recursos, entre outros. Problemas de otimização combinatória podem apresentar métodos exatos e eficientes de resolução, porém alguns destes problemas têm a necessidade de métodos não exatos (heurísticos), uma vez que sua formulação ou sua resolução exata levaria a uma complexidade intratável ou indesejável. Em todos os casos, a resolução de problemas de otimização combinatória pode ser feita usando alguma linguagem de notação matemática, como *Optimization Programming Language* (OPL), *Algebraic Modeling Programming Language* (AMPL) e *General Algebraic Modeling System* (GAMS).

A implementação dos métodos para resolução de problemas de otimização combinatória pode utilizar algoritmos paralelos, aproveitando melhor os recursos computacionais. Mas, para usar algoritmos paralelos de forma mais efetiva deve-se ter máquinas com o maior número de processadores possível.

Neste contexto, tem-se a *General Purpose Graphics Processing Unit* (GPGPU) ou unidade de processamento gráfico de propósito geral. Uma GPGPU inclui o uso de uma *Graphics Processing Unit* (GPU), ou unidade de processamento gráfico, juntamente com uma *Central Processing Unit* (CPU), ou unidade de processamento central, para acelerar aplicações de uso geral, científicas e de engenharia. A combinação de CPU e GPU é poderosa porque CPUs consistem de alguns núcleos otimizados para processamento serial, enquanto GPUs consistem em milhares de núcleos menores, mais eficientes, projetados para desempenho paralelo (NVIDIA CORPORATION, 2012a). Assim, partes seriais do código são executadas na CPU, enquanto partes paralelas executam na GPU.

Diante do exposto, desenvolveu-se uma ferramenta para interpretar *scripts* OPL, gerando código paralelo para máquina com placa de vídeo GPU. O código será gerado em C/C++.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é desenvolver uma ferramenta para interpretar *scripts* escritos na linguagem OPL, gerando código paralelo para máquinas com placa de vídeo.

Os objetivos específicos do trabalho são:

a) fazer as análises léxica, sintática e semântica dos *scripts* OPL, mostrando erros de compilação;
b) traduzir os *scripts* OPL para código em C/C++, independente de plataforma;
c) disponibilizar a implementação paralela em C/C++ para máquinas com placas de vídeo, de pelo menos um método de resolução de problemas de otimização combinatória para ser utilizado pelos códigos em C/C++ gerados.

1.2 ESTRUTURA DO TRABALHO

O presente trabalho foi dividido em quatro capítulos, sendo que neste capítulo foram apresentados a introdução e os objetivos do trabalho.

O capítulo 2 apresenta a fundamentação teórica, iniciando com tradutores de linguagens. Em seguida é descrita a linguagem OPL e a ferramenta IBM ILOG CPLEX. Ainda no capítulo 2 são apresentados os algoritmos de otimização combinatória, tecnologia GPU e trabalhos correlatos.

O terceiro capítulo relata o desenvolvimento da ferramenta proposta, iniciando com os requisitos, seguidos da especificação, da implementação e da operacionalidade da mesma. Também são apresentados os resultados obtidos.

O último capítulo traz as conclusões do desenvolvimento deste trabalho bem como alguns aspectos que ficaram em aberto, servindo de sugestões para futuras extensões.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo está organizado em cinco seções. A primeira seção trata de tradutores de linguagens. A próxima seção apresenta a linguagem OPL, bem como a ferramenta para elaboração e execução de *scripts* OPL, a IBM ILOG CPLEX. A terceira seção trata de algoritmos para resolver problemas de otimização combinatória. A seção 2.4 apresenta a tecnologia GPU CUDA. Por fim, a última seção descreve trabalhos correlatos à ferramenta desenvolvida.

2.1 TRADUTORES DE LINGUAGENS

Tradutores de linguagens, segundo Aho et al. (2008, p. 3), mapeiam um programa fonte para um programa objeto semanticamente equivalente. Tradutores podem ser classificados em interpretadores e compiladores. Um compilador constitui-se de etapas com funções específicas.

A etapa inicial de um compilador é a análise léxica, onde ocorre a leitura do programa fonte, caracter a caracter, com a identificação de *tokens*, isso é, dos menores elementos de uma linguagem, tais como símbolos especiais, palavras reservadas, identificadores, entre outros. Grune et al. (2001, p. 54) explica que os *tokens* podem ser definidos com expressões regulares. O Quadro 1 mostra a definição do *token* `número`.

Quadro 1 – Exemplo de *token*

```
numero = digito digito*
digito = 0|1|2|3|4|5|6|7|8|9
```

Fonte: Louden (2004, p. 97).

A próxima etapa de um compilador consiste em identificar sequências de *tokens* que formam as estruturas sintáticas da linguagem. Esta fase chama-se análise sintática e tem como resultado uma estrutura do tipo árvore com a representação intermediária da estrutura gramatical da sequência de *tokens* proveniente da analise léxica.

A sintaxe de uma linguagem é especificada pelas regras gramaticais de uma gramática, normalmente descrita usando a notação *Backus-Naur Form* (BNF). O Quadro 2 apresenta uma BNF que define as regras gramaticais de uma expressão aritmética, composta por 7 regras, usando o *token* `numero`, descrito no Quadro 1.

Quadro 2 – Exemplo de BNF

```
<exp> ::= <exp> <op> <exp> | (<exp>) | numero
<op> ::= + | - | * | /
```

Fonte: Louden (2004, p.99).

A análise semântica tem como função principal utilizar a árvore criada na análise sintática para validar se a estrutura do programa fonte está de acordo com a definição

semântica da linguagem. A definição da semântica de uma linguagem é o conjunto de restrições contextuais da linguagem, tais como: declaração e uso de identificadores e compatibilidade de tipos (WATT; BROWN, 2000, p. 136).

Não existe um método padrão para especificar a semântica. Um método geralmente utilizado pelos desenvolvedores de compiladores para descrever a semântica das linguagens de programação é identificar atributos, ou propriedades, de entidades da linguagem e escrever regras semânticas, que expressem como esses atributos se relacionam com as regras gramaticais da linguagem. Este conjunto de atributos é denominado gramática de atributos. São mais utilizados em linguagens que seguem o princípio de semântica dirigida pela sintaxe, onde o conteúdo semântico está fortemente relacionado com a sintaxe (LOUDEN, 2004, p. 260). Se as regras semânticas especificarem ações com efeitos colaterais, tem-se os chamados esquemas de tradução (PRICE; TOSCANI, 2001, p. 86).

A geração de código é feita transformando a árvore , resultante da análise sintática, em segmento de código. O código gerado pode ser um código intermediário, como notação polonesa ou código de 3 endereços, código objeto final ou código em outra linguagem de programação (PRICE; TOSCANI, 2001, p. 115).

2.2 OPL

Conforme IBM Software (2011a), OPL é uma linguagem de modelagem que acelera o desenvolvimento e a implementação de métodos de otimização combinatória usando programação linear/quadrática baseada em restrições. OPL é uma linguagem de modelagem fácil e possui ferramentas intuitivas para teste, definição de perfil e ajuste dos modelos.

Um exemplo de *script* em OPL para escolha de produção em uma fábrica de gás pode ser visto no Quadro 3.

Quadro 3 - *Script* em OPL para escolha de produção (nome do *script*: `gas.mod`)

```
1   {string} Products = ...;
2   {string} Components = ...;
3
4   float Demand[Products][Components] = ...;
5   float Profit[Products] = ...;
6   float Stock[Components] = ...;
7   dvar float+ Production[Products];
8
9   maximize
10    sum( p in Products )
11      Profit[p] * Production[p];
12
13  subject to {
14    forall( c in Components )
15      ct:
16        sum( p in Products )
17          Demand[p][c] * Production[p] <= Stock[c];
18  }
```

Fonte: IBM Software (2007).

No Quadro 3 tem-se que:

a) nas linhas 1 e 2 são declarados os vetores (de `string`) `Products` e `Components`;

b) na linha 4 é declarada a matriz `Demand` com tamanho relativo ao tamanho de `Products` e `Components`;

c) nas linhas 5 e 6 são declarados os vetores (de `float`) `Profit` e `Stock`;

d) na linha 7 é declarado o vetor (`float+`) `Production`, que é a variável de decisão;

e) as linhas 9 a 11 definem a função objetivo, neste caso, maximizar o lucro;

f) as linhas 13 a 18 contêm a restrição para evitar ultrapassar o limite de capacidade em estoque;

g) atribuições com valor `...` indicam que o valor correspondente encontra-se em um arquivo de dados.

Como entrada de dados para um *script* pode ser usado um arquivo texto, banco de dados ou planilha do Excel. Para o *script* do Quadro 3 pode ser utilizado um arquivo texto com extensão `.dat` (do Quadro 4).

Quadro 4 - Entrada de dados para o *script* do Quadro 3

```
1   Products = { "gas" "chloride" };
2   Components = { "nitrogen" "hydrogen" "chlorine" };
3
4   Demand = [ [4 3 2] [3 6 3] ];
5   Profit = [30 40];
6   Stock = [30 50 40];
```

Fonte: IBM Software (2007).

Na resolução dos problemas modelados pode-se utilizar a ferramenta IBM ILOG CPLEX, que é a ferramenta oficial para execução de *scripts* OPL. Segundo IBM Software (2007), a ferramenta tem um conjunto de algoritmos implementados internamente para

resolver problemas de otimização combinatória, tais como Simplex, Dual Simplex, Barrier e Sifting, permitindo a escolha do algoritmo que melhor se enquadra na resolução do problema.

O desenvolvimento de programas em OPL é facilitado com o ambiente de desenvolvimento IBM ILOG CPLEX *Optimization Studio* baseado no Eclipse, com suporte a depuração e testes dos programas. Pode-se integrar programas de terceiros, escritos em diversas linguagens de alto nível como C++, C# e Java, com *scripts* OPL. Para fins de teste, é disponibilizada uma versão da ferramenta IBM ILOG CPLEX sem custo, com algumas limitações no tamanho do *script* desenvolvido e com licença de 90 dias.

2.3 ALGORITMOS DE OTIMIZAÇÃO COMBINATÓRIA

Programaçao linear é o conjunto de técnicas que permite resolver problemas de otimização combinatória (TAVARES; CORREIA, 1999, p. 213). O desenvolvimento da programação linear foi inspirado em pelo menos três tipo de problemas:

a) transporte: otimizar a distribuição de mercadorias para cinco armazéns localizados em diferentes cidades (Nova York, Chicago, Kansas, Dallas, São Francisco) a partir de três unidades localizadas em outras três cidades (Portland, Seatle, San Diego), sabendo os custos de transporte, a demanda de cada armazém e a capacidade máxima de produção de cada unidade;
b) composição: otimizar a composição de uma dieta para minimizar o custo satisfazendo níveis mínimos de calorias, a partir do preço e dos conteúdos vitamínicos e calóricos de diversos alimentos;
c) formação e produção: determinar qual o programa ótimo de contratação e de formação para realizar a produção de uma encomenda em um determinado prazo. A indústria deve aumentar o número de operários especializados, contratando novos funcionários, e usar parte dos funcionários atuais para efetuar o treinamento e a formação dos novos operários.

Para solucionar problemas de otimização combinatória existem diversos algoritmos, cada um adequado para resolver determinados tipos de problemas. Alguns algoritmos, como o Simplex ou o Dual Simplex, encontram a solução exata de um problema. Em algumas situações a solução demora muito tempo para ser encontrada. Nestas situações, segundo Barr et al. (1995, p. 10), pode-se usar alguma técnica de solução aproximada, como *Greedy Randomized Adaptive Search Procedure* (GRASP) ou Tabu *search*.

O algoritmo Tabu *search*, conforme Piwakowski (1996, p. 1-2), dada uma função objetivo e um conjunto de possíveis soluções, a partir de alguma solução inicial escolhida

arbitrariamente, explora o conjunto das possíveis soluções até que a primeira que satisfaça as condições do problema seja encontrada.

> [O algoritmo GRASP] é um método iterativo, onde cada iteração é composta basicamente por uma etapa de construção seguida de uma etapa de busca local de uma solução. Este par de operações é então repetido um determinado número de vezes. Na etapa de construção, iterativamente se define elemento a elemento de uma solução através de uma lista de candidatos (LC) previamente ordenados segundo sua aptidão. Dentre estes candidatos, seleciona-se um candidato aleatoriamente. [...] A solução obtida pelo GRASP na sua fase de construção não é necessariamente um ótimo local, portanto é sempre aconselhável o uso de técnicas de busca local para tentar melhorar a qualidade da atual solução dentro de uma estrutura de vizinhança pré-determinada. A melhor solução obtida até o momento é armazenada e retorna-se à etapa de construção até que um critério de parada seja acionado. (OCHI; DRUMMOND; SILVA, 2000, p. 357).

Já o algoritmo Simplex, utilizado neste trabalho, foi o primeiro algoritmo desenvolvido para resolução de problemas de otimização combinatória (DANTZIG, 1998, p. 120). Desenvolvido em 1947 por George Dantzig, consiste em encontrar uma solução básica inicial e ir sucessivamente buscando outras soluções básicas viáveis, acarretando melhores valores para a função objetivo até ser obtida a solução ótima.

Segundo Lalami, Boyer e El-Baz (2011, p. 2001), para a resolução do algoritmo Simplex em ambiente computacional é criada uma matriz denominada tableu. O tableu tem n linhas, onde n é igual ao número de restrições do problema mais 1, e tem n + m + 1 colunas, onde m é o número de variáveis. A primeira linha dessa matriz é preenchida com os coeficientes da função objetivo com o sinal negativo. Para cada restrição é preenchido o seu coeficiente na linha subsequente. Na última coluna do tableu é preenchido o valor limite da restrição. O bloco bidimensional vazio é preenchido com uma matriz identidade.

Após a montagem do tableu, o algoritmo consiste em:

a) encontrar o menor valor negativo na primeira linha, chamado coluna pivô;
b) dividir cada valor da última coluna pelo valor correspondente na coluna pivô, ignorando a primeira linha;
c) encontrar o índice do valor de menor resultado positivo, esse será a variável de saída;
d) atualizar a linha da variável de saída usando o índice encontrado no passo anterior;
e) atualizar o tableu ignorando a linha da variável de saída;
f) atualizar a coluna da variável de saída;
g) se existirem valores negativos na primeira linha, então voltar ao primeiro passo, caso contrário, imprimir a solução encontrada.

Alguns algoritmos de otimização combinatória, como o Simplex, GRASP ou Tabu *search* podem ser desenvolvidos para a versão paralela, obtendo aproveitamento de recursos

computacionais (IBM SOFTWARE, 2007; LALAMI; BOYER; EL-BAZ, 2011; MOREIRA; MENESES, 2012).

2.4 TECNOLOGIA GPU

Em novembro de 2006 a NVIDIA Corporation (2012b) introduziu no mercado a tecnologia CUDA[1], uma tecnologia de processamento gráfico para computação de propósito geral que aproveita a computação paralela em computadores com placa de vídeo NVIDIA para solucionar problemas computacionais de modo mais eficiente do que em CPU.

Conforme NVIDIA Corporation (2012a), a tecnologia ganhou há alguns anos a atenção de cientistas da computação que usam computação de alta performance. Devido à grande quantidade de processadores disponíveis em uma GPU, a capacidade de processamento total supera ou iguala a processadores comuns, apesar de serem inferiores em potência. GPU tem a vantagem de estar evoluindo tecnologicamente mais rápido do que processadores centrais. Sendo assim, com o crescente interesse da comunidade científica nas placas gráficas, a NVIDIA Corporation (2012b) tornou a GPU totalmente programável. A programação é baseada na arquitetura *Single Instruction Multiple Data* (SIMD) em que a execução é feita paralelamente em vários processadores, cada um deles processando uma parte dos dados.

Pode-se desenvolver programas em C ou C++, usando o CUDA *Toolkit* 5.0, que contém: um compilador – o NVIDIA's CUDA *Compiler* (NVCC), bibliotecas matemáticas e uma *Integrated Development Environment* (IDE) – Nsight Eclipse. O *toolkit* é disponibilizado para os sistemas operacionais Linux, Mac OS e Windows, sendo que para desenvolvimento em Windows é necessário o *plugin* do Visual Studio disponível dentro do CUDA *Toolkit*.

O modelo de programação CUDA assume que *threads* CUDA são a menor unidade de execução e estas executam paralelamente o mesmo código sobre diferentes dados. As *threads* são agrupadas em blocos e os blocos são agrupados em *grids*. Essa característica abstrai do programador a divisão física de execução e transfere essa decisão à GPU. Para obter vantagem na utilização de GPU para processamento, o problema precisa conter um processamento intenso de dados que possa ser paralelizado, fazendo com que esses códigos sejam executados em GPU e a parte sequencial do programa continue executando na CPU,

[1] A tecnologia *Compute Unified Device Architecture* (CUDA), uma das tecnologias mais utilizadas atualmente para obter proveito de processamento paralelo, foi criada pela NVIDIA Corporation, empresa fabricante de placas de vídeo (NVIDIA CORPORATION, 2012a).

tornando, deste modo, a GPU um co-processador da CPU. O modelo também especifica que a placa de vídeo tenha sua própria área física de memória denominada memória *device*.

Threads CUDA podem acessar dados a partir múltiplos espaços de memória durante a execução. Cada *thread* tem sua própria memória privada local, sendo que cada bloco de *threads* tem um bloco de memória compartilhada entre todas as *threads* desse bloco. Um nível superior de memória é a memória global. Existe ainda dois espaços adicionais de memória na placa de vídeo: memória constante e memória de textura, sendo apenas para leitura e acessíveis por todas as *threads*. A memória de textura como o nome sugere permite o armazenamento de texturas e tem otimizações para carregamento de texturas e endereçamento de memória.

CUDA C é a extensão da linguagem C para utilização de recursos CUDA. Permite o programador declarar funções C chamadas `kernels`. Uma função `kernel` é definida usando a declaração `__global__` antes da definição da função. Para fazer uma chamada a uma função `kernel` deve-se usar o nome da função seguido por `<<<número de grids, número de threads por grid>>>` conforme pode ser visto no Quadro 5.

Quadro 5 – Exemplo de código CUDA C

```
1   // kernel definition
2   __global__ void VecAdd(float* A, float* B, float* C) {
3     int i = threadIdx.x;
4     C[i] = A[i] + B[i];
5   }
6
7   int main() {
8     // invocação de função Kernel com N threads
9     VecAdd<<<1, N>>>(A, B, C);
10  }
```

Fonte: NVIDIA Corporation (2012a).

Quando a função `VecAdd` for chamada, ela será executada `N` vezes em paralelo por diferentes *threads*. Nesse código, cada uma das `N` *threads* vai executar uma soma entre os vetores A e B e armazenar o valor em C.

2.5 TRABALHOS CORRELATOS

Esta seção descreve alguns trabalhos correlatos, incluindo o trabalho de Gesser (2007), um compilador para código Java 5.0 que gera código C++ para plataforma Palm OS; o trabalho de Lalami, Boyer e El-Baz (2011), que descreve a implementação do método Simplex usando a tecnologia CUDA e o trabalho de Moreira e Meneses (2012), que apresenta o algoritmo Tabu *search* usando CUDA.

2.5.1 Compilador Java 5.0

Gesser (2007) desenvolveu um compilador para analisar código Java 5.0, gerando como saída o código correspondente em C++ nativo para a plataforma Palm OS. A motivação do trabalho teve origem na dificuldade de implementar código fonte diretamente nas linguagens suportadas pelo Palm OS.

Foi utilizado o JavaCC para geração dos analisadores léxico e sintático, além da *Abstract Syntax Tree* (AST). Foi necessária a implementação em C++ de algumas bibliotecas Java, como `java.util` e `java.awt`, além de um mecanismo para simular o gerenciamento de memória como o *garbage colector* do Java. Como teste de validação do trabalho, o autor desenvolveu um jogo tradicional do Palm OS em linguagem Java e, posteriormente, usando o compilador gerou o programa equivalente em C++.

2.5.2 Método Simplex em um sistema CPU-GPU

Lalami, Boyer e El-Baz (2011), da Universidade de Toulese na França, desenvolveram uma versão do método Simplex para execução híbrida entre CPU e GPU. Foi utilizada a tecnologia CUDA para o desenvolvimento do algoritmo. Para tentar diminuir qualquer sobrecarga, a solução foi implementada sem utilizar as bibliotecas matemáticas tradicionais cuBLAS e *Linear Algebra PACKage* (LAPACK). A implementação utiliza valores numéricos de precisão dupla para manter a qualidade dos resultados.

Como máquina de teste foi utilizada uma placa gráfica GTX 260. Utilizando dados aleatórios para testes, foi percebida uma melhora no tempo de execução de 12.5 vezes. Segundo Lalami, Boyer e El-Baz (2011, p. 2005), os resultados obtidos foram superiores à execução sequencial, provando ser viável a utilização de placas de vídeo em otimização combinatória.

2.5.3 Algoritmo Tabu search em GPUs

Moreira e Meneses (2012) apresentam a implementação do algoritmo Tabu *search* de forma paralela para execução em placas de vídeo GPU. Os testes foram realizados em um computador Dell Precision T3500 com a seguinte configuração: processador Intel Xeon 64 bits; 4 núcleos de 2.53 GHz; 5.8 GB de memória RAM; GPU FX 1800 (1.3 GHz, 64 bits, 8 multi-processadores, 64 núcleos, 768 MB de memória global) e sistema operacional Linux Ubuntu 11.04, 32 bits.

“As soluções ótimas das instâncias testadas foram obtidas por meio do software IBM ILOG CPLEX versão 12.4” (MOREIRA; MENESES, 2012). Executando o algoritmo com

poucas variáveis de entrada, a implementação sequencial foi mais rápida que a versão paralela. Isso se deve ao custo de comunicação entre a CPU e a GPU. Já para instâncias maiores, o benefício do paralelismo foi proporcional ao número de variáveis das instâncias do problema. Os autores relataram também a facilidade em usar unidades de processamento gráfico e a arquitetura CUDA para implementar algoritmos paralelos.

3 DESENVOLVIMENTO

Neste capítulo são apresentadas as etapas de desenvolvimento do trabalho fundamentado no estudo realizado e apresentado no capítulo anterior. Encontram-se definidos os requisitos da ferramenta, seguidos da especificação, da implementação e da descrição dos resultados obtidos.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

A ferramenta proposta deve:

a) analisar *scripts* OPL, baseados na ferramenta IBM ILOG CPLEX, reportando erros léxicos, sintáticos e semânticos (Requisito Funcional – RF01);
b) usar a implementação paralela em C++ para GPU do algoritmo Simplex (RF02);
c) gerar código C++ como código intermediário do processo de compilação (RF03);
d) usar algum compilador CUDA para geração de código executável a partir do código intermediário gerado (RF04);
e) ser implementado em C++, utilizando o ambiente de desenvolvimento CUDA *Toolkit* 5.0 (Requisito Não Funcional – RNF01).
f) gerar código C/C++ independente de plataforma (Requisito Não Funcional - (RNF02).

3.2 ESPECIFICAÇÃO

Nesta seção é apresentada a especificação da ferramenta, iniciando com as construções gramaticais da linguagem OPL e as gramáticas definidas para analisá-la. Também são descritos os diagramas de casos de uso, de classes e de sequência da *Unified Modeling Language* (UML), os quais foram criados utilizando a ferramenta Enterprise Architect (EA).

3.2.1 Linguagem OPL

A linguagem OPL é especificada por duas gramáticas distintas. A primeira é a gramática de *scripts*, que contém as regras gramaticais da lógica do programa. A segunda gramática é a de dados, relativamente mais simples, responsável por possibilitar a entrada de dados dos *scripts* OPL.

Na especificação de uma linguagem de programação, deve-se definir os símbolos léxicos, as regras gramaticais e a semântica. Sendo assim, o Quadro 6 mostra os *tokens* utilizados no *script* OPL mostrado no Quadro 3 (capítulo anterior).

Quadro 6 – *Tokens* OPL

token	descrição
`maximize`	palavra reservada usada para identificar que o problema é de maximização
`sum`	palavra reservada usada para identificar o somatório
`in`	palavra reservada usada para identificar operação em uma lista
`subject`	palavra reservada usada para identificar restrições, deve ser seguida por `to`
`to`	palavra reservada usada para identificar restrições, deve ser precedida por *`subject`*
`forall`	palavra reservada usada para identificar uma iteração em uma lista
`dvar`	palavra reservada usada para identificar uma variável de decisão
`float`	palavra reservada usada na declaração de ponto flutuante
`string`	palavra reservada usada na declaração de *string*
`Id`	*token* usado para descrever o padrão de formação identificadores
símbolos especiais	`{ } = ... ; [ ] ( ) : + * <=`

Um *script* OPL é essencialmente dividido em três blocos, o primeiro é a declaração de variavéis, o segundo a definição da função objetivo e o terceiro são as restrições. Em função da extensão da gramática de *scripts* OPL, o Quadro 7 exibe apenas as construções sintáticas usadas na definição da função objetivo do *script* OPL do Quadro 3. Os *tokens* e a gramática completa encontram-se em IBM Software (2011b).

Quadro 7 – Construções sintáticas de *scripts* OPL

```
...
Objective: ... | "maximize" Expression

Expression: ... | PathExpression | BinaryExpression | AggregateExpression | ...

PathExpression: LocationExpression | ArraySlotExpression | ...
LocationExpression: Id | ...
ArraySlotExpression: PathExpression '[' Expression ']' | ...

BinaryExpression: ... | Expression "*" Expression

AggregateExpression: "sum" '(' Qualifiers ')' Expression | ...
Qualifiers: Qualifier | Qualifiers ',' Qualifier
Qualifier: SimpleQualifier | ...
SimpleQualifier: Pattern "in" Expression Filter_opt
Pattern: Expression
Filter_opt: /* empty */ | ...
...
```

Para a ferramenta desenvolvida, as definições semânticas da linguagem não foram alteradas. No entanto, para atender ao requisito funcional RF03, é gerado código em C++ para algumas construções sintáticas da linguagem. A correspondência de código para declaração de variável externa em OPL e C++, definição da função objetivo e restrições encontram-se, respectivamente, nos Quadro 8, Quadro 9 e Quadro 10.

Quadro 8 – Código em OPL x código em C++ para declaração externa de variável

construção OPL	construção C++
`{string} Products = ...;`	`extern vector<string> Products;`
`float Demand[Products][Components] = ...;`	`double Demand DemandVal;`
`float Profit[Products] = ...;`	`double Profit ProfitVal;`

Quadro 9 – Código em OPL x código em C++ para declaração de função objetivo

construção OPL	construção C++

`maximize`
`  sum( p in Products )`
`  Profit[p] * Production[p];`
	`sizeArray =`
	`sizeof(Profit)/sizeof(Profit[0]);`
	`for(int i = 0; i < sizeArray; i++){`
	`  obj.valores.push_back(Profit[i]);`
	`}`

Quadro 10 – Código em OPL x código em C++ para declaração de restrição

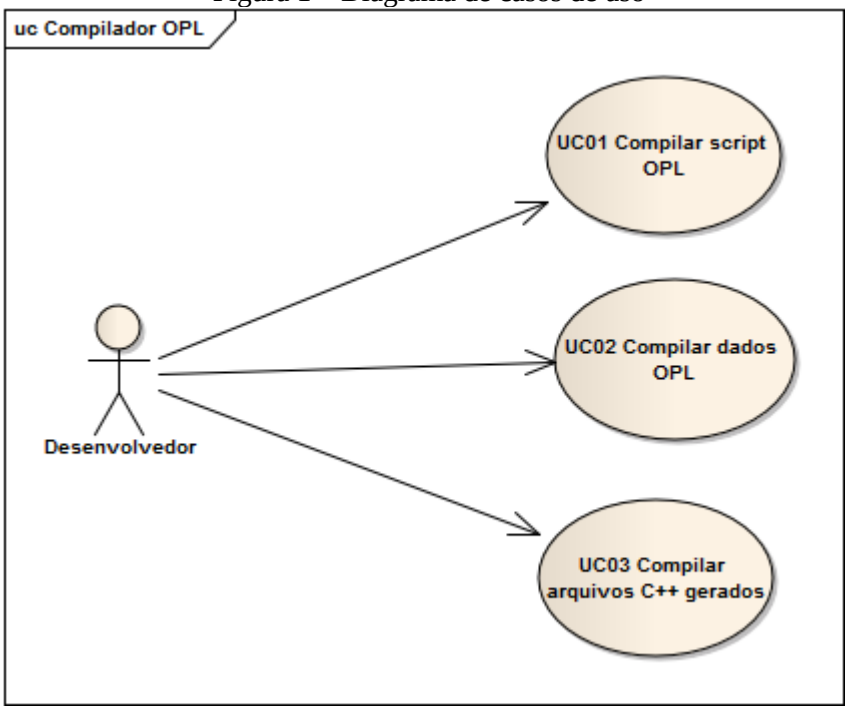
construção OPL	construção C++

`subject to {`
` forall( c in Components )`
`  ct:`
`  sum( p in Products )`
`  Demand[p][c] * Production[p]`
`  <= Stock[c];`
`}`
	`for(int c=0; c < Components.size(); c++) {`
	`  Restricao _restricao;`
	`  _restricao.id = "ct";`
	`  for(int p = 0; p < Products.size(); p++){`
	`  _restricao.pesos.push_back(Demand[p][c]);`
	`  }`
	`  _restricao.limite = Stock[p];`
	`  restricoes.push_back(_restricao);`
	`}`

3.2.2 Casos de uso

A ferramenta é composta por três casos de usos, visualizados na Figura 1 e detalhados a seguir.

Figura 1 – Diagrama de casos de uso



O `UC01 - Compilar script OPL` (Quadro 11) é responsável por compilar o arquivo de *script* OPL.

Quadro 11 – Detalhamento do caso de uso `Compilar *script* OPL`

UC01 - Compilar *script* OPL	
Descrição	Verificar se *scripts* OPL possuem erros léxicos, sintáticos ou semânticos e gerar arquivos C++ equivalentes.
Pré-condições	Existir um *script* OPL, arquivo com a extensão `.mod`.
Fluxo principal	1. O desenvolvedor executa o compilador de *scripts*, informando como parâmetro o arquivo fonte OPL que deve ser compilado. 2. O compilador executa as análises léxica, sintática e semântica do arquivo de entrada. 3. O compilador gera código C++.
Fluxo de exceção	Caso ocorra erro em algum passo do fluxo principal, o compilador exibe os erros detectados.
Pós-condições	Código C++ gerado em arquivo `Cplex.cpp`.

O `UC02 - Compilar dados OPL` (Quadro 12) é responsável por compilar o arquivo de dados OPL.

Quadro 12 – Detalhamento do caso de uso `Compilar dados OPL`

UC02 - Compilar dados OPL	
Descrição	Verificar se os arquivos de dados OPL possuem erros léxicos, sintáticos ou semânticos e gerar arquivos C++ equivalentes.
Pré-condições	Existir um arquivo de dados OPL com a extensão `.dat`.
Fluxo principal	1. O desenvolvedor executa o compilador de arquivos de dados, informando como parâmetro o arquivo fonte OPL que deve ser compilado. 2. O compilador executa as análises léxica, sintática e semântica do arquivo de entrada. 3. O compilador gera código C++.
Fluxo de exceção	Caso ocorra erro em algum passo do fluxo principal, o compilador exibe os erros detectados.
Pós-condições	Código C++ gerado em arquivo `Cplex.h`.

O caso de uso `UC03 – Compilar arquivos C++ gerados` faz parte do escopo da proposta do trabalho, mas não da ferramenta em si. Com base nos arquivos gerados com a ferramenta, o arquivo com a implementação do algoritmo Simplex e os arquivos auxiliares, pode-se realizar a compilação do projeto com qualquer versão do compilador NVCC. O caso de uso está detalhado no Quadro 13.

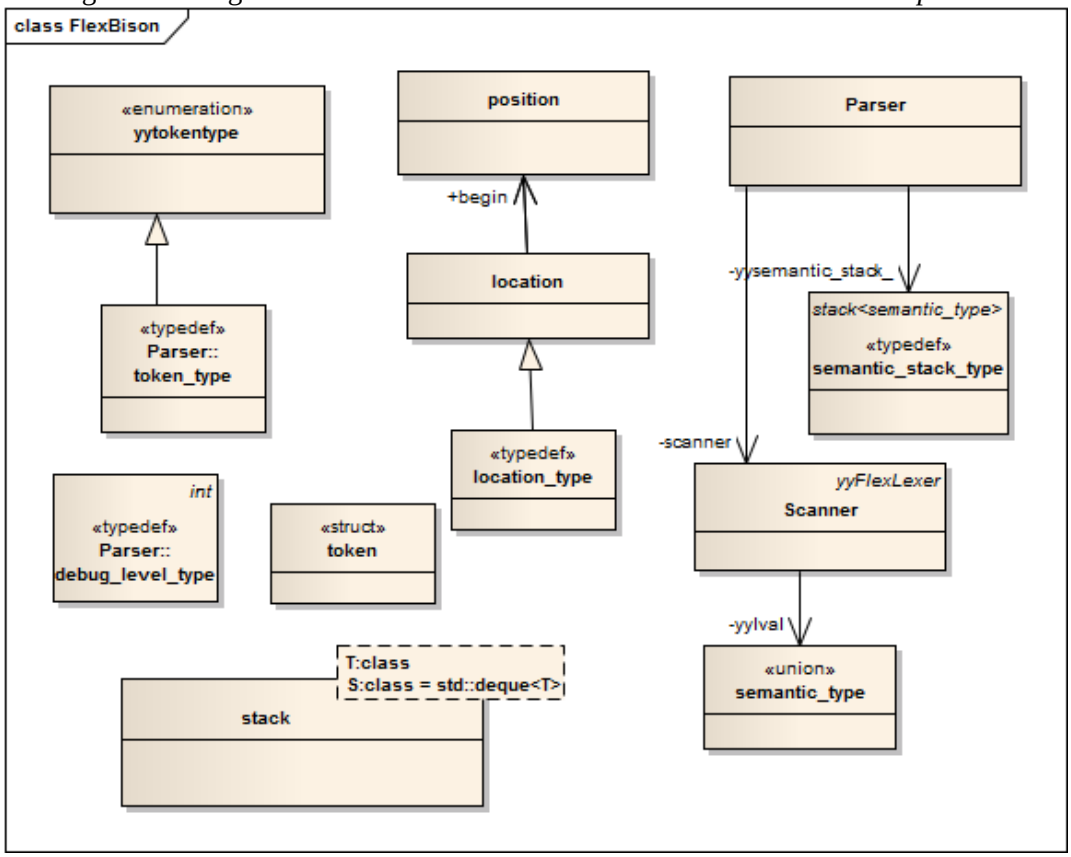
Quadro 13 – Detalhamento do caso de uso Compilar arquivos C++ gerados

UC03 - Compilar arquivos C++ gerados	
Descrição	Gerar um programa executável nativo para NVIDIA CUDA.
Pré-condições	Existir um projeto CUDA composto pelos arquivos `Cplex.cpp` e `Cplex.h`, gerados pela ferramenta, além do arquivo `TesteCuda.cu` e arquivos auxiliares.
Fluxo principal	1. O desenvolvedor executa o compilador NVCC, informando os arquivos fonte C++. 2. O compilador executa as análises léxica, sintática e semântica dos arquivos de entrada. 3. O compilador gera código executável nativo para NVIDIA CUDA.
Fluxo de exceção	Caso ocorra erro em algum passo do fluxo principal, o compilador exibe erros do programa.
Pós-condições	Código executável nativo gerado para placa de vídeo.

3.2.3 Diagrama de classes

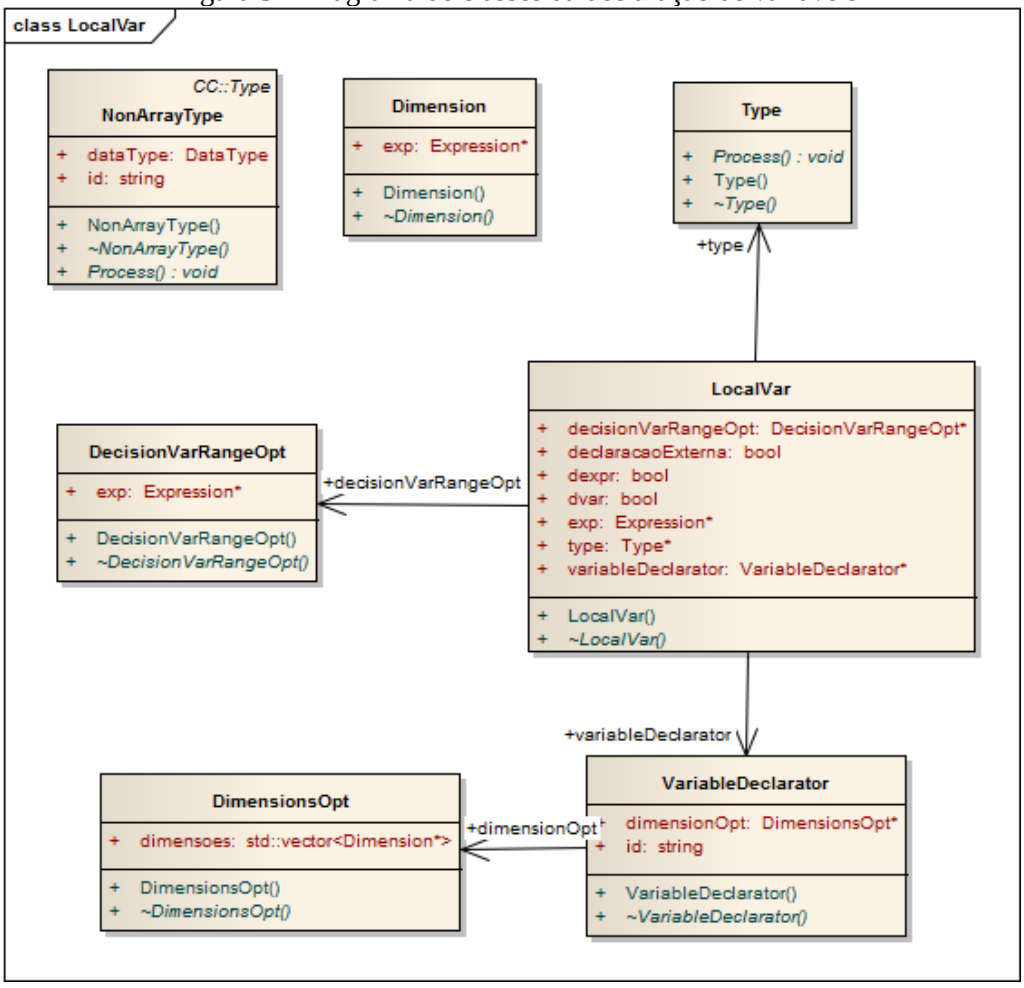
Devido a grande quantidade de classes que compõem a especificação da ferramenta, elas foram agrupadas por funcionalidade. Na Figura 2 são mostradas as classes específicas do compilador de *scripts* OPL. Todas as classes deste diagrama foram geradas usando as ferramentas Flex e Bison. Por isso, optou-se por apresentar as classes sem a reprensentação dos atributos e dos métodos. No diagrama, a classe `Scanner` e a enumeração `yytokentype` representam o mapeamento léxico, sendo a primeira responsável pelo processamento e a segunda apenas utilizada para enumerar os possíveis *tokens*. As outras classes representam o mapeamento sintático do programa, tendo como classe principal `Parser`, que utiliza as demais para identificar e retornar a construção sintática identificada.

Figura 2 – Diagrama de classes dos analisadores léxico e sintático de *scripts* OPL



A Figura 3 contêm o diagrama de classes para o conjunto de classes que representa a declaração de variáveis e o Quadro 14 traz a descrição de cada classe.

Figura 3 – Diagrama de classes da declaração de variáveis

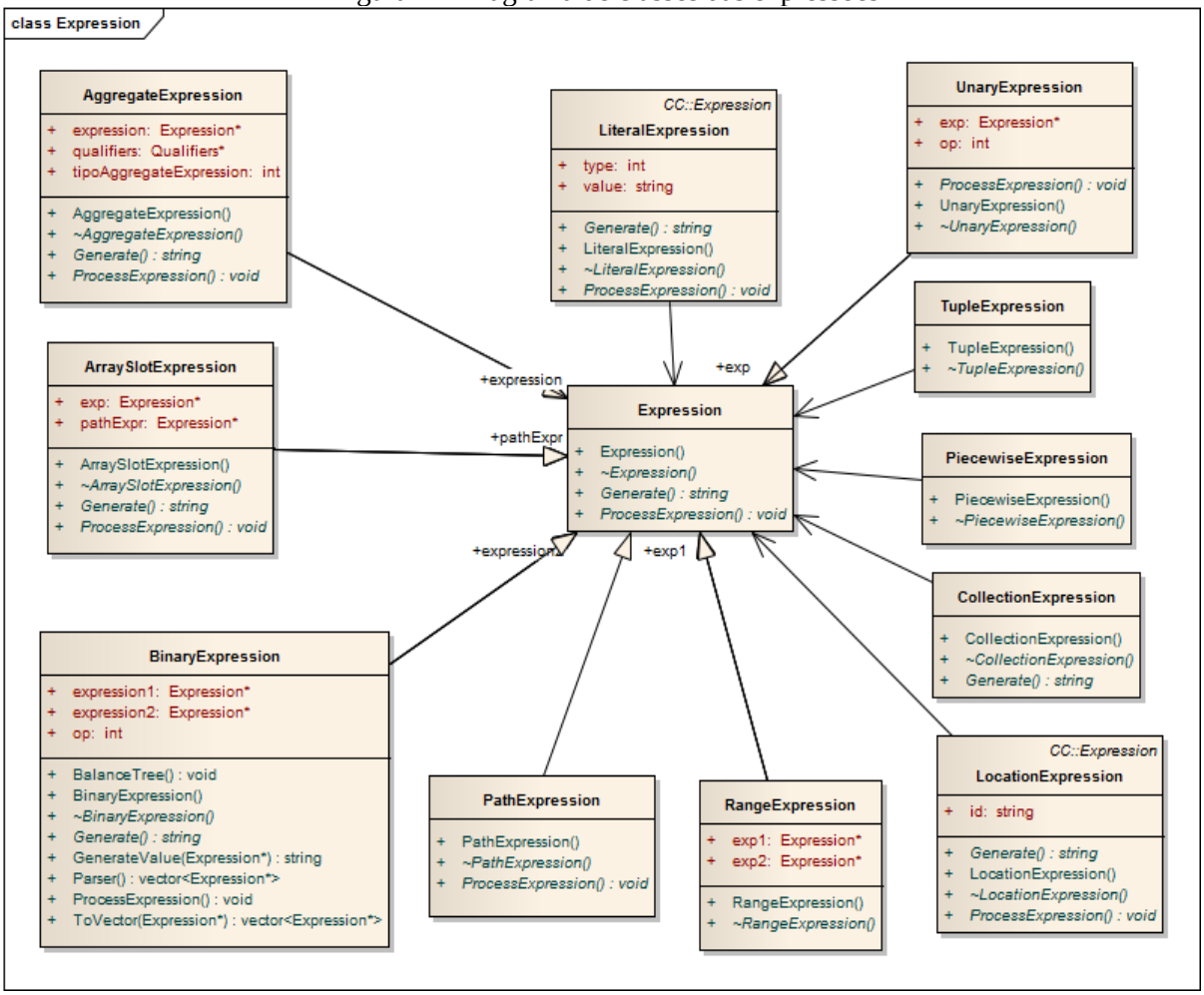


Quadro 14 – Descrição das classes da declaração de variáveis

classe	descrição
`DecisionVarRangeOpt`	mapeia variáveis estruturadas, como listas, especificamente para variáveis de decisão
`Dimension`	mapeia variáveis estruturadas, como `arrays`
`DimensionOpt`	mapeia variáveis estruturadas, como listas
`LocalVar`	classe base para o mapeamento de variáveis
`NonArrayType`	mapeia variáveis primitivas, como `int` ou `float`
`Type`	mapeia o tipo de dado das variáveis
`VariableDeclarator`	mapeia variáveis com mais do que uma dimensão

Na Figura 4 está o diagrama de classes que representa as expressões, as quais são responsáveis por tratar a função objetivo de um *script* OPL. No Quadro 15 é listada a descrição de cada classe.

Figura 4 – Diagrama de classes das expressões



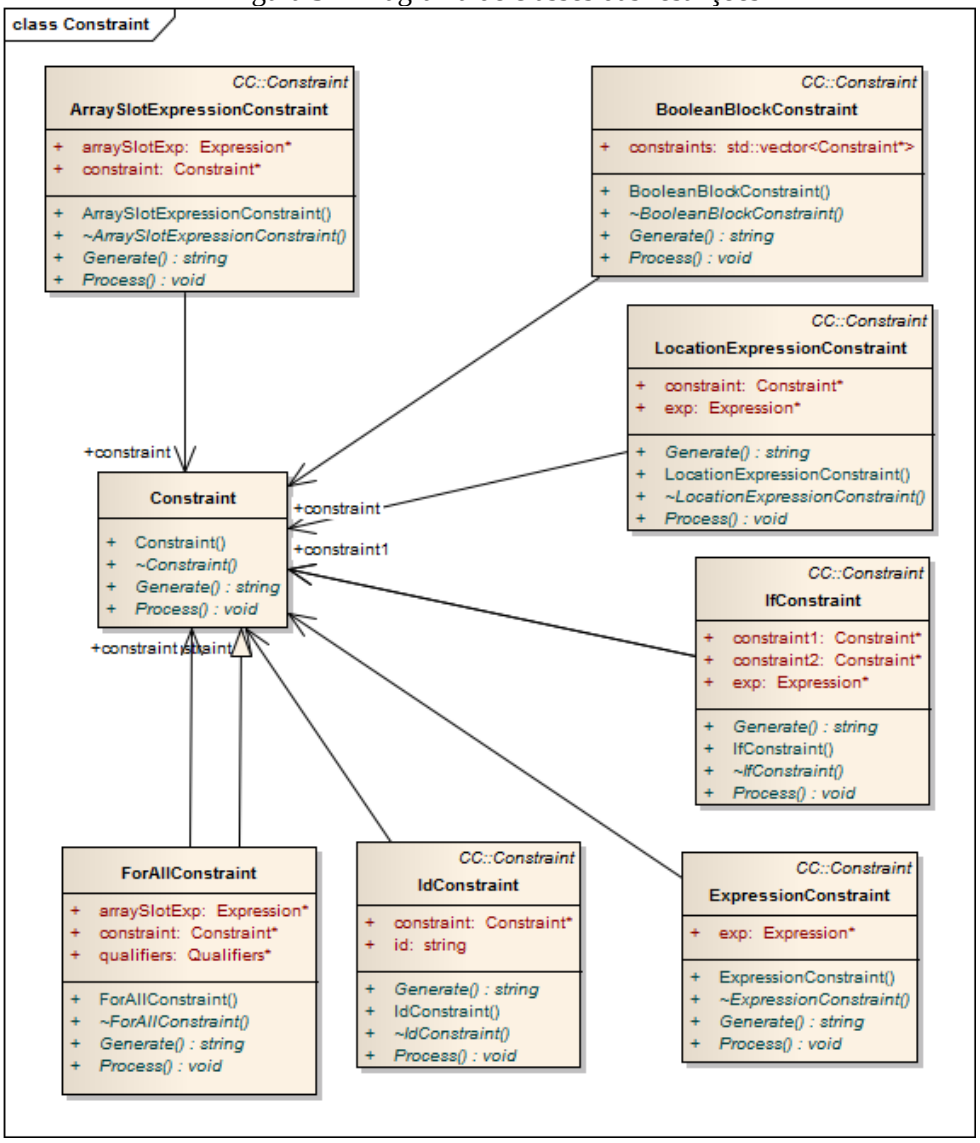
Quadro 15 – Descrição das classes das expressões

classe	Descrição
`AggregateExpression`	mapeia a função `sum`, um qualificador e uma expressão
`ArraySlotExpression`	mapeia o acesso a dados em `array`
`BinaryExpression`	mapeia as expressões binárias
`Expression`	classe base para expressões
`LiteralExpression`	mapeia expressão literais, contendo o valor e tipo de dado
`LocationExpression`	mapeia expressão contendo uma variável
`PathExpression`	mapeia expressão contendo uma variável ou subscrito
`RangeExpression`	mapeia as expressões com intervalo de valores
`UnaryExpression`	mapeia as expressões com sinal unário (+ ou -)

As classes `CollectionExpression`, `PiecewiseExpression` e `TupleExpression` foram especificadas para mapear as construções correspondentes em OPL. Porém, a ferramenta desenvolvida não suporta essas construções na análise semântica e geração de código.

As classes que representa as restrições de um *script* OPL são exibidos na Figura 5 e descritas no Quadro 16.

Figura 5 – Diagrama de classes das restrições



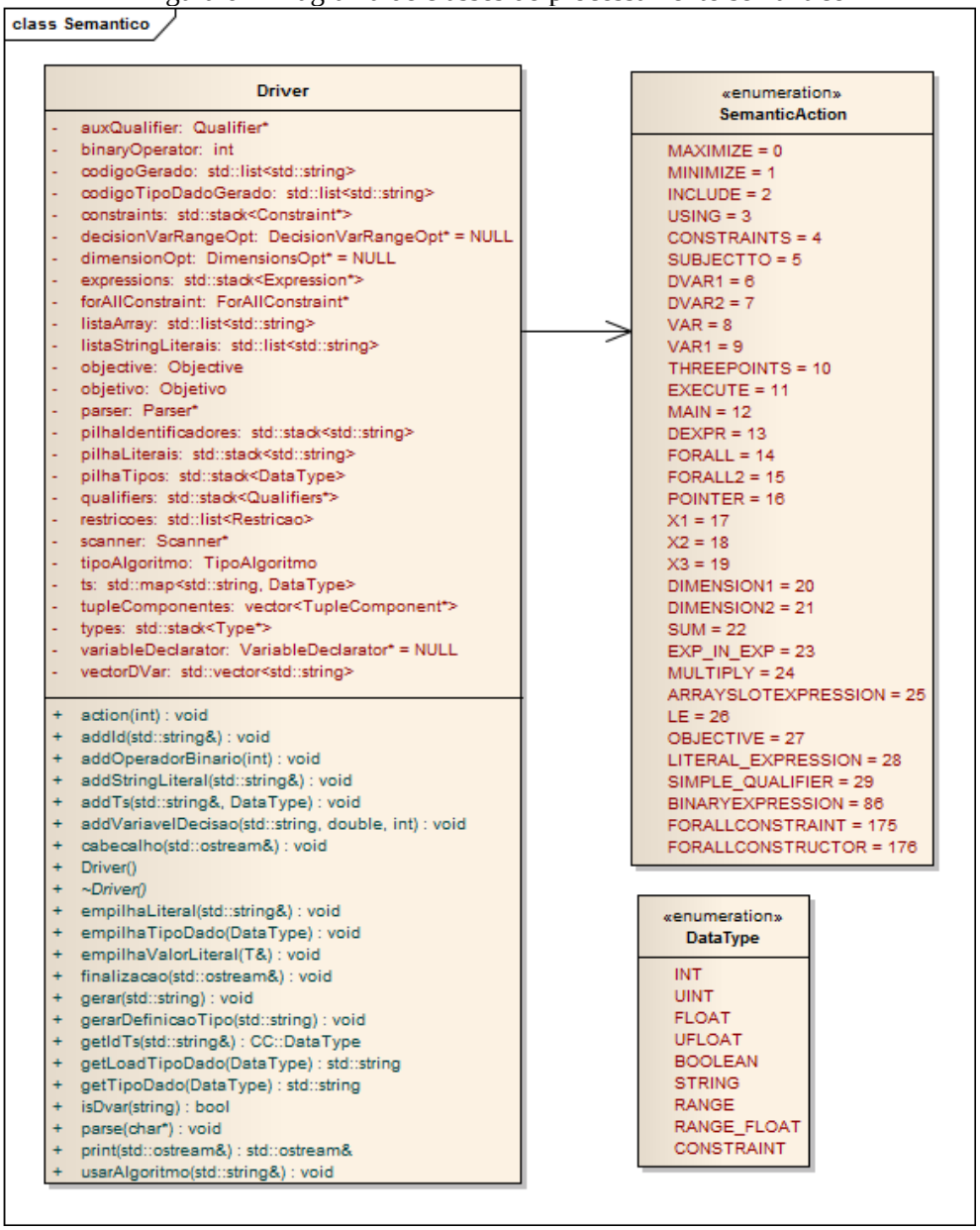
Quadro 16 – Descrição das classes das restrições

classe	descrição
`Constraint`	classe base para mapear restrições
`ExpressionConstraint`	mapeia restrição aplicada a uma expressão
`ForAllConstraint`	mapeia restrição aplicada a uma lista
`IdConstraint`	mapeia restrição rotulada

As classes `ArraySlotExpressionConstraint`, `BooleanBlockConstraint`, `IfConstraint` e `LocationExpressionConstraint` foram especificadas, mas não foram tratadas semanticamente.

As classes responsáveis pela análise semântica são exibidas na Figura 6 e descritas no Quadro 17.

Figura 6 – Diagrama de classes do processamento semântico

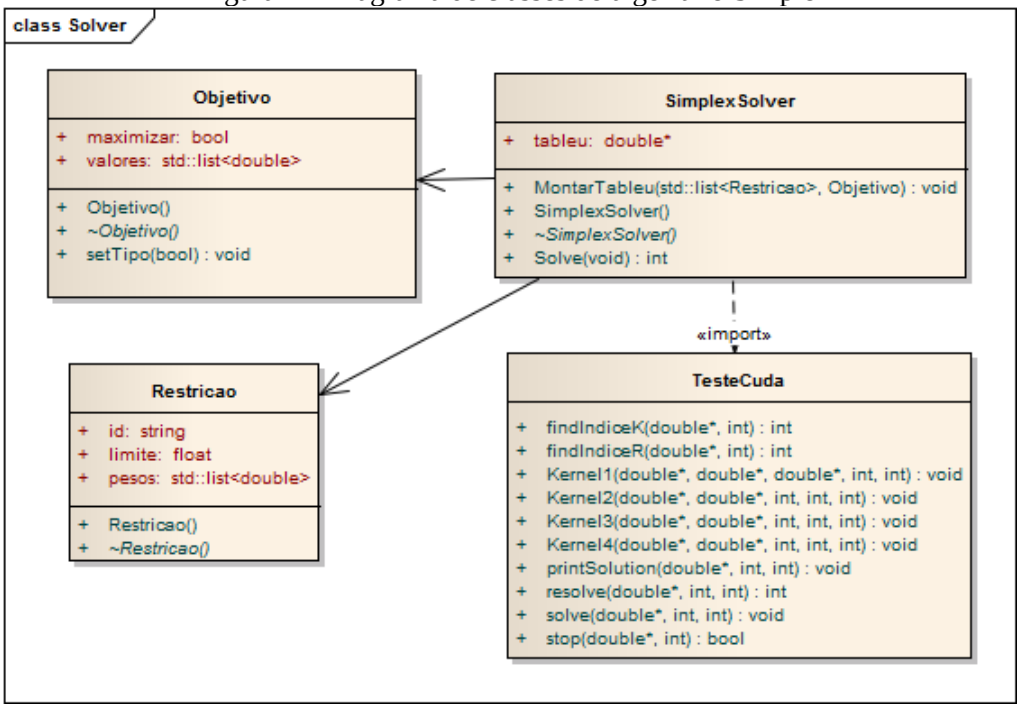


Quadro 17 – Descrição das classes do processamento semântico

classe	descrição
`DataType`	representa os tipos de dados
`Driver`	realiza o processamento semântico e gera o código `Cplex.cpp`
`SemanticAction`	representa as possíveis ações semânticas

As classes responsáveis pelo algoritmo Simplex são exibidas na Figura 7 e descritas no Quadro 18.

Figura 7 – Diagrama de classes do algoritmo Simplex

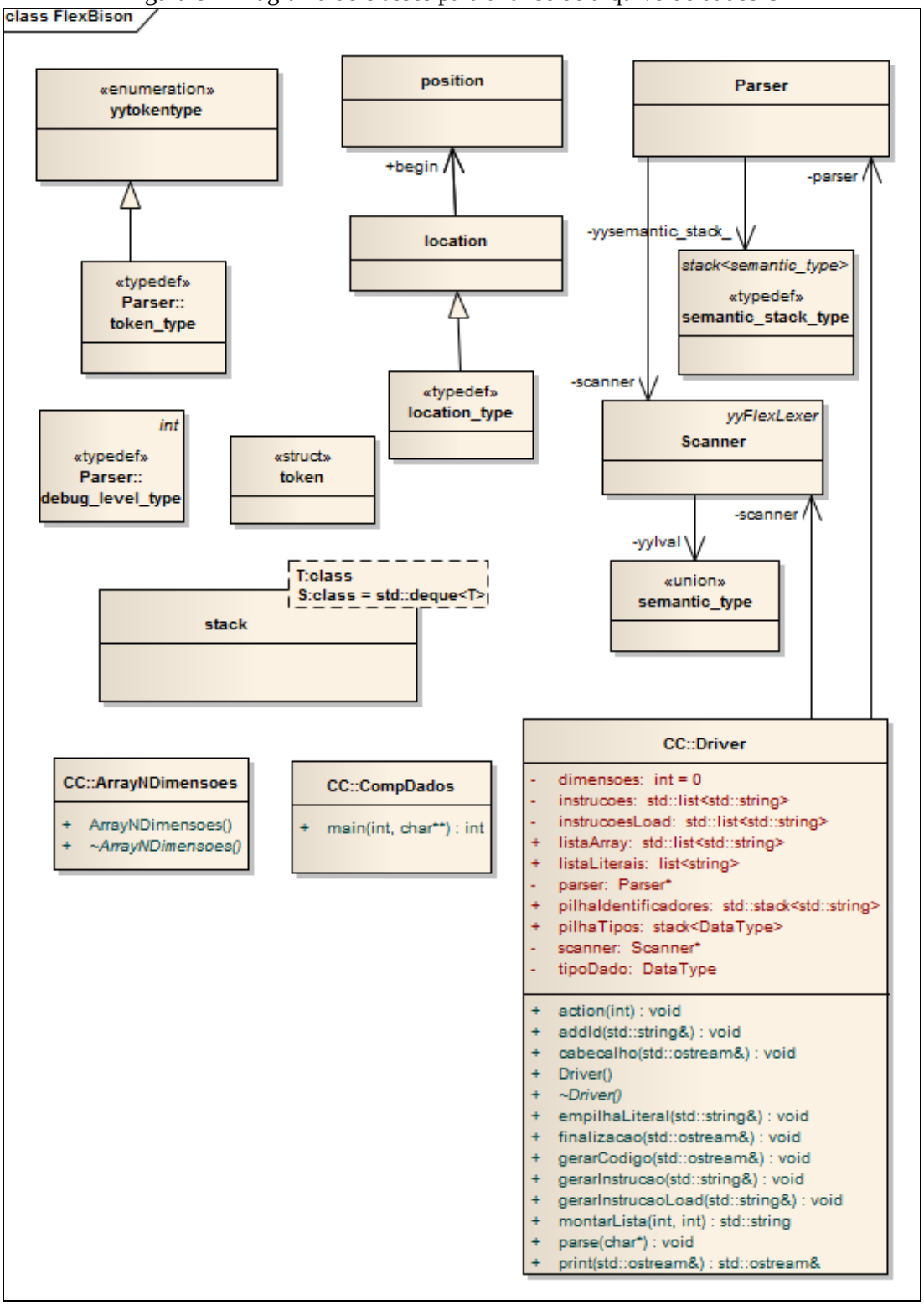


Quadro 18 – Descrição das classes do algoritmo Simplex

classe	descrição
`Objetivo`	representa a função objetivo
`Restricao`	representa as restrições do problema
`SimplexSolver`	monta o tableu com base nas restrições e função objetivo, fazendo a interface com a classe `TesteCuda`
`TesteCuda`	representa a implementação do algoritmo Simplex

As classes que representam as análises léxica, sintática e semântica para os arquivos de dados OPL estão na Figura 8 e suas descrições no Quadro 19. As classes que não foram relacionadas no Quadro 19 são classes auxiliares criadas automaticamente com o Flex e Bison.

Figura 8 – Diagrama de classes para análise de arquivo de dados OPL



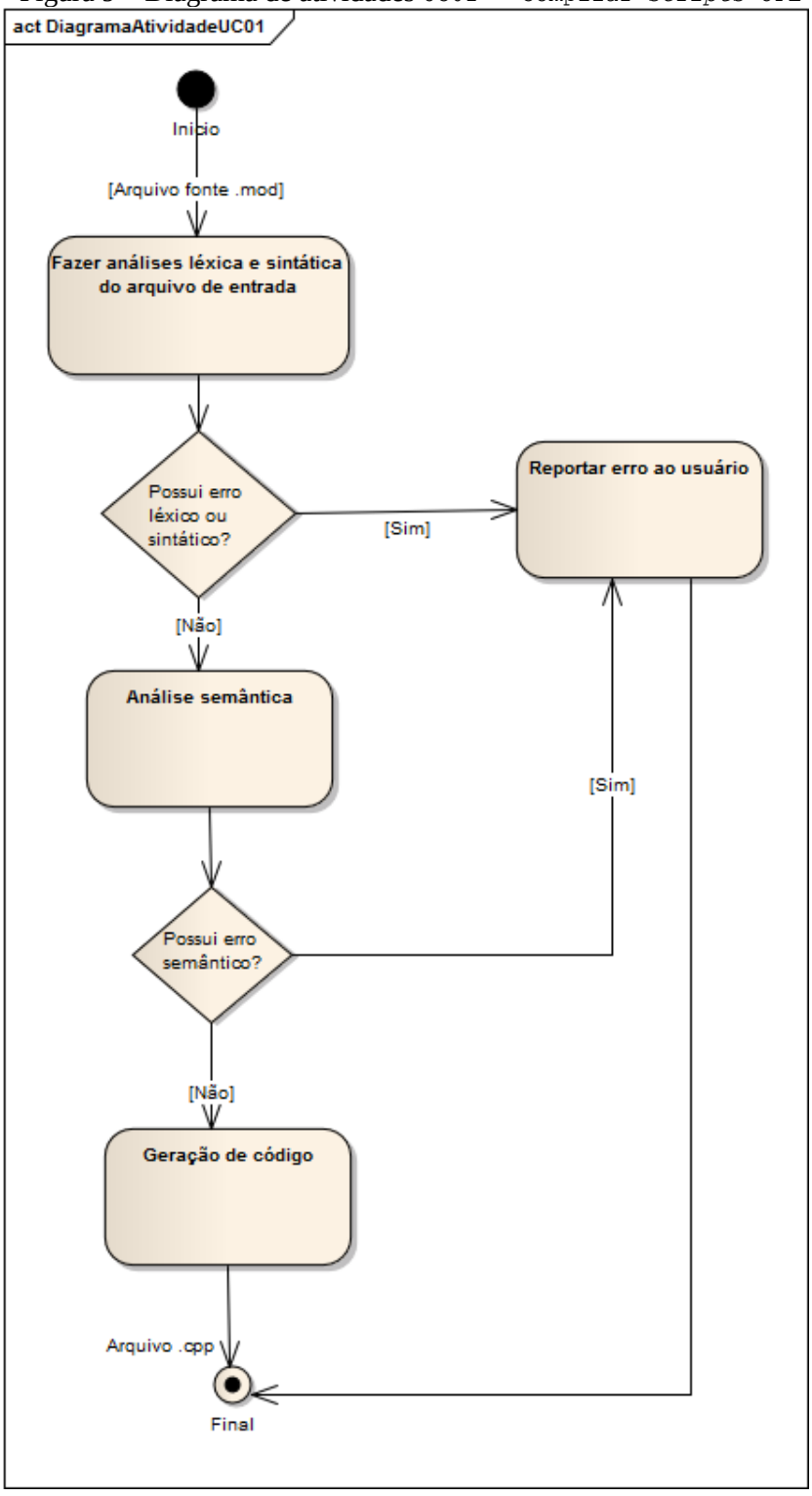
Quadro 19 – Descrição das classes para análise de arquivos de dados OPL

classe	descrição
`ArrayNDimensoes`	armazena informações da definição de `array`
`CompDados`	arquivo inicial do programa, recebe como parâmetro o caminho do arquivo de dados OPL a ser compilado.
`Driver`	processa o arquivo de dados OPL e gera o código `Cplex.h` com a entrada de dados

3.2.4 Diagrama de atividades

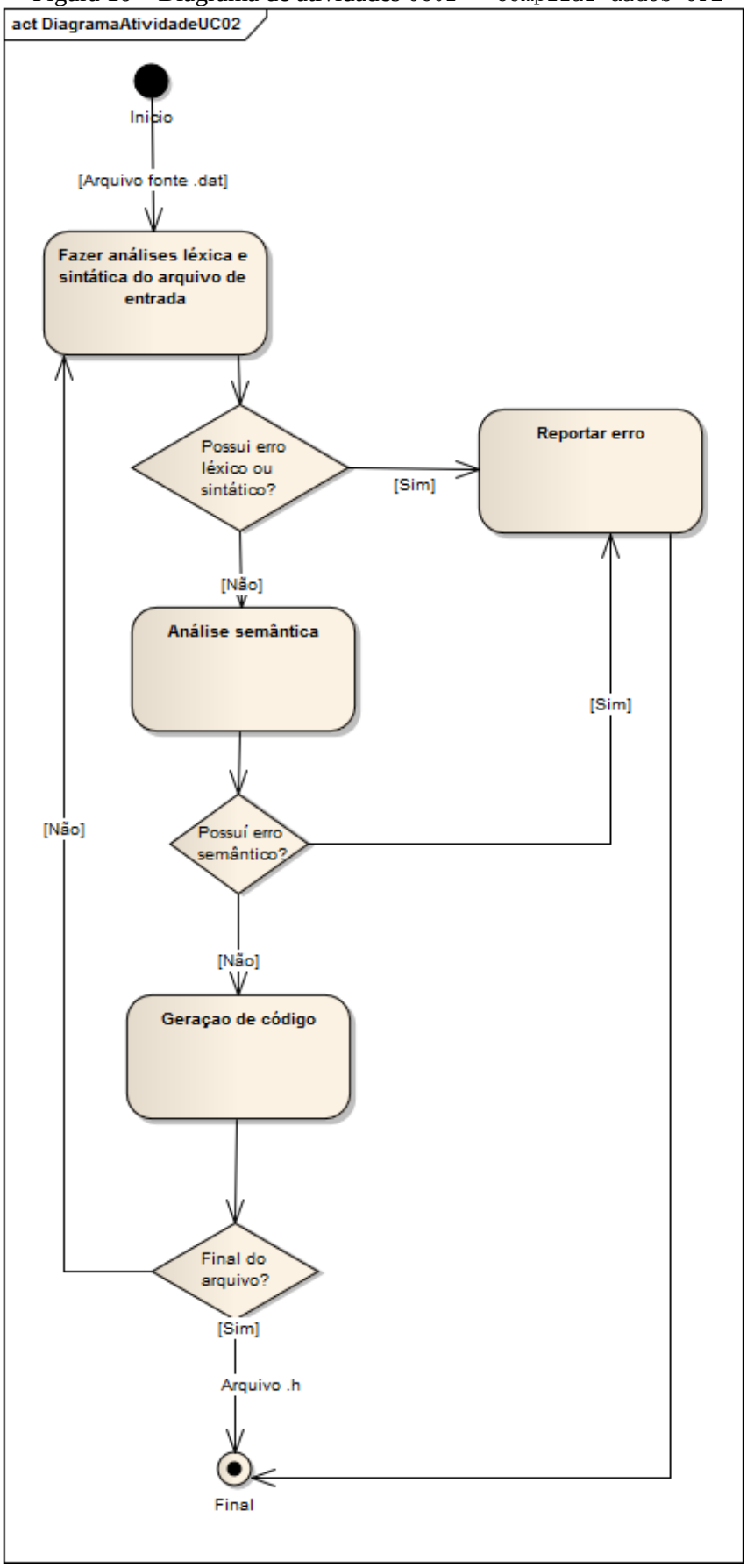
As atividades executadas pelo caso de uso `UC01 - Compilar script OPL` são detalhadas no diagrama apresentado na Figura 9. Tendo como entrada um *script* OPL (arquivo com extensão `.mod`), são realizas as análises léxica e sintática. Caso seja detectado algum erro, o mesmo é reportado e o processo finalizado. Caso contrário, é realizada a análise semântica. Da mesma forma, em caso de erro, o mesmo é reportado ao usuário e a compilação é encerrada. Caso o *script* OPL não possua erros semânticos, é efetuada a geração do código C++ correspondente em um arquivo com extensão `.cpp`.

Figura 9 – Diagrama de atividades `UC01 - Compilar scripts OPL`



O caso de uso `UC02 - Compilar dados OPL` trata da compilação dos arquivos com dados OPL. Na Figura 10 é mostrado o diagrama de atividades correspondente. Tendo como entrada um arquivo de dados OPL (arquivo com extensão `.dat`), também são realizadas as análises léxica, sintática, semântica e a geração de código. No decorrer do fluxo, qualquer erro encontrado será reportado ao usuário e o processo é encerrado.

Figura 10 – Diagrama de atividades `UC02 - Compilar dados OPL`



3.3 IMPLEMENTAÇÃO

Nessa seção é detalhada a implementação da ferramenta, começando com as técnicas e ferramentas utilizadas, seguidas do compilador de dados e de *script* OPL, do projeto utilizado como base para testes de execução e do algoritmo Simplex. Também é descrita a operacionalidade da implementação.

3.3.1 Técnicas e ferramentas utilizadas

O protótipo foi escrito na linguagem C++ utilizando o ambiente de programação Nsight Eclipse. Os analisadores léxicos, tanto para os *scripts* como para os dados OPL, foram gerados usando a ferramenta Flex, enquanto os analisadores sintáticos foram gerados utilizando a ferramenta Bison. No desenvolvimento do código C++ do algoritmo Simplex foi utilizado o CUDA *Toolkit* 5.0, usando técnicas para alocação de memória em GPU e processamento com múltiplas *threads*. Para compilar os arquivos com código C++ gerados pela ferramenta, assim como o arquivo fonte com a implementação do algoritmo Simplex, foi utilizado o compilador NVCC.

3.3.2 Compiladores para *scripts* e dados OPL

Foram implementados dois compiladores, um para os *scripts* e outro para os dados. A especificação léxica dos compiladores foi feita com base na especificação das gramáticas da linguagem OPL (IBM SOFTWARE, 2011b). A construção dos analisadores léxicos foi automatizada usando a ferramenta Flex, que é a versão C++ do já conhecido Lex. A ferramenta tem como entrada um arquivo com a extensão `.l`, contendo as expressões regulares, que definem os *tokens*, e código C++. Como resultado do processamento da especificação léxica, o Flex gera o arquivo `lex.yy.cc` em C++. A especificação léxica de *scripts* e dados encontra-se, respectivamente, nos Apêndices A e C.

Os analisadores sintáticos foram desenvolvidos usando a ferramenta Bison. A entrada para a ferramenta Bison é um arquivo com a extensão `.yy`, contendo a especificação da gramática, onde são definidas quais construções a linguagem suporta, podendo também ser adicionado código C++. As gramáticas de *scripts* e dados da OPL encontram-se nos Apêndices B e D.

Neste trabalho foi usado o método de tradução dirigida pela sintaxe, onde o analisador sintático chama o analisador léxico sempre que precisar de um *token*, através do método `yylex()`, que retorna um inteiro representando o *token* encontrado, e chama o analisador

semântico, através do método `action`, responsável pela análise semântica do programa. Durante o processo, o analisador sintático constroe uma AST com os *tokens* encontrados.

A análise semântica é feita a partir do método `action` da classe `Driver`, chamado pelo analisador sintático. A maioria das ações semânticas mapeadas são ações auxiliares para a montagem da AST correspondente. No entanto, determinadas ações semânticas tratam toda a árvore gerada até então, podendo gerar o código final. O código final gerado é salvo em um arquivo chamado `Cplex.cpp` dentro do projeto `TesteCuda`.

No Quadro 20 é apresentado o código que faz a análise e a geração de código para a declaração de variável externa. Para possibilitar que variáveis tenham valor inicial declarado no arquivo de dados, foi utilizado o recurso do C++ de declaração externa de variável e macros `#DEFINE`. Na linha 2 é criado um objeto `LocalVar` que é utilizado para verificação semântica. As linhas 3 e 4 preenchem esse objeto com informações da declaração de variável obtidas no fonte. As linhas 7 e 8 verificam se a construção é válida, enquanto as linhas 10, 11, 13, 14 e 15 fazem a geração do código objeto.

Quadro 20 – Análise e geração de código para declaração externa de variável

```
1   case THREEPOINTS: {
2     LocalVar localVar;
3     localVar.variableDeclarator = variableDeclarator;
4     localVar.type = types.top();
5     types.pop();
6     string codigo;
7     if(dynamic_cast<NonArrayType*>(localVar.type)) {
8       NonArrayType *n = dynamic_cast<NonArrayType*>(localVar.type);
9       if(n->dataType == STRING)
10        codigo = "extern vector<string> "
11                 + localVar.variableDeclarator->id + ";\n";
12      else {
13        codigo = getTipoDado(n->dataType) + " "
14                 + localVar.variableDeclarator->id + " "
15                 + localVar.variableDeclarator->id + "Val;\n";
16      }
17      gerar(codigo);
18    }
19    else
20      cout << "Falha" << endl;
21    break;
22  }
```

No Quadro 21 é apresentado o código que faz a análise e a geração de código para declaração da função objetivo. As linhas 1 a 15 fazem a verificação semântica desta construção e as linhas 16 a 22 fazem a geração do código objeto.

Quadro 21 – Análise e geração de código para declaração função objetivo

```
1   string AggregateExpression::Generate() {
2   switch (tipoAggregateExpression) {
3     case 0: //sum
4     {
5     if(expression == NULL || qualifiers == NULL){
6       return "Erro ao gerar AggregateExpression";
7     }
8     ret = "//Objetivo\n";
9     BinaryExpression *bExp = dynamic_cast<BinaryExpression*>(expression);
10    if(bExp != NULL) {
11      vector<Expression*> vectorBinaryExp = bExp->Parser();
12      for(int i = 0; i < vectorBinaryExp.size(); i++) {
13        bExp = dynamic_cast<BinaryExpression*>(vectorBinaryExp.at(i));
14        ArraySlotExpression *aExp =
15                              dynamic_cast<ArraySlotExpression*>(bExp->expression1);
16        LocationExpression *lExp =
17                                 dynamic_cast<LocationExpression*>(aExp->pathExpr);
18        ret += "sizeArray = sizeof("+lExp->id+") / sizeof("
19               + lExp->id + "[0]);\n";
20        ret += "for(int i = 0; i < sizeArray; i++)\n{\n";
21        ret += "   obj.valores.push_back("+lExp->id + "[i]);\n";
22        ret += "}\n";
23      }
24    }
25    break;
26    }
27    default:
28    break;
29  }
30  return ret;
31  }
```

No Quadro 22 é apresentado o código que faz a análise e a geração de código para declaração de restrições. A verificação semântica de uma restrição `ForAllConstrait` é feita nas linhas 1 a 6, a geração de código objeto é feita nas linhas 8 a 13. A verificação semântica de uma restrição `ExpressConstraint` é feita nas linhas 23 a 28, enquanto a geração de código é feita nas linhas 30 a 35.

Quadro 22 – Análise e geração de código para declaração restrição

```
1   string ForAllConstraint::Generate() {
2     string codigo = "//Restricoes\n";
3     BinaryExpression *bExp = dynamic_cast<BinaryExpression*>
4                                    (qualifiers->stackQualifier.top()->expression);
5     if(bExp != NULL) {
6       if(bExp->op == 14)//IN
7       {
8         string exp1 = bExp->expression1->Generate();
9         codigo += "for(int "+ exp1 + "= 0;" + exp1 + " < " +
10                   bExp->expression2->Generate()+ ".size(); " + exp1 + "++)\n{\n";
11        codigo += "Restricao _restricao;\n";
12        codigo += constraint->Generate();
13        codigo += "}\n";
14      }
15      else
16        return NULL;
17    }
18    return codigo;
19  }
20
21  string ExpressionConstraint::Generate() {
22    string codigo = "";
23    AggregateExpression *agrE = dynamic_cast<AggregateExpression*>(exp);
24    if (agrE != NULL) {
25      if (agrE->tipoAggregateExpression == 0) //sum
26      {
27        BinaryExpression *bExp = dynamic_cast<BinaryExpression*>
28                                 (agrE->qualifiers->stackQualifier.top()->expression);
29        if (bExp != NULL) {
30          codigo += "for(int " + bExp->expression1->Generate() + " = 0; " +
31                    bExp->expression1->Generate() + " < " +
32                    bExp->expression2->Generate() + ".size(); " +
33                    bExp->expression1->Generate() + "++)\n{\n";
34          codigo += agrE->expression->Generate();
35          return codigo;
36        }
37      }
38    }
39    else
40      return "Falha";
41  }
```

3.3.3 Projeto base para teste

Para fins de teste do código gerado, foi desenvolvido um projeto CUDA denominado `TesteCuda`, que contém os arquivos fontes gerados, além de alguns arquivos fonte auxiliares. Os arquivos auxiliares são `Objetivo.h` e `Restricao.h`, cujos códigos encontram-se nos Apêndices E e F, e são usados por `SimplexSolver.cpp` para montar a entrada do algoritmo Simplex. O arquivo `SimplexSolver.cpp` faz também interface com o código fonte CUDA do arquivo `TesteCuda.cu`, conforme o Quadro 23.

Quadro 23 – `SimplexSolver.h`

```
1   #include "SimplexSolver.h"
2   extern "C" int resolve(double *tableu, int linhas, int colunas);
3
4   namespace CC {
5   void SimplexSolver::MontarTableu(vector<Restricao> restricoes, Objetivo
6   objetivo){
7     cout << "restricoes =" << restricoes.size();
8     cout << " objetivos =" << objetivo.valores.size() << endl;
9     int linhas = restricoes.size() +1;
10    int colunas = objetivo.valores.size() + linhas;
11    tableu = new double[linhas * colunas];
12
13    linhasCount = linhas;
14    colunasCount = colunas;
15
16    //preenche objetivo
17    for(int i = 0; i < objetivo.valores.size(); i++) {
18      tableu[i+1] = -objetivo.valores[i];
10    }
20
21    //preenche restricoes
22    for(unsigned int i = 0; i < restricoes.size(); i++) {
23      Restricao r = restricoes[i];
24      //ignora primeira linha do tableu
25      tableu[(i+1) * colunas] = r.limite;
26      for(int j = 0; j < r.pesos.size(); j++) {
27         tableu[(i+1) * colunas + 1 + j] = r.pesos[j];
28      }
29    }
30
31    //preenche identidade
32    for(unsigned int i = 1; i < linhas; i++) {
33      int indice = i*colunas + linhas +(i+1);
34      tableu[indice] = 1;
35    }
36  }
37
38  int SimplexSolver::Solve(void) {
39    return resolve(tableu, linhasCount, colunasCount);
40  }
41  } /* namespace CC */
```

Na linha 11 do quadro anterior, é criado o tableu com base na quantidade de restrições e objetivos. Na linha 18 é feito o preenchimento da função objetivo. Nas linhas 22 a 29 é preenchido o tableu com as informações de restrições. Na linha 2 é feito o *link* com a função externa `resolve`, declarada no arquivo `TesteCuda.cu`.

3.3.4 Algoritmo Simplex

Neste trabalho foi feita a implementação da forma padrão do algoritmo Simplex, onde todas as restrições devem ser do tipo “menor igual”.

O algoritmo tem como entrada um ponteiro de tipo `double`, que representa o tableu, isto é, a estrutura contendo os dados do problema formatado. Nesta implementação é alocado um `array` de tamanho `N x M`, onde: `N` é a quantidade de restrições mais 1, `M` é a quantidade de valores da função objetivo mais `N`. A primeira linha do tableu é preenchida com o valor de cada variável da função objetivo com sinal negativo. A primeira coluna é preenchida com os

valores limites de cada restrição, sendo que a primeira coluna da primeira linha não é preenchida. Em seguida, os valores das restrições são copiados para as linhas e as colunas internas ao tableu. O tableu é alocado em memória RAM do computador e precisa ser copiado para a memória da placa de vídeo (memória do *device*).

O código é dividido em etapas, conforme os passos do algoritmo Simplex apresentado na seção 2.3, sendo que as executadas na CPU são denominadas *host* e as executadas na GPU são chamadas *device*. Os métodos com a diretiva `__global__` são executados na GPU.

O primeiro passo do Simplex consiste em encontrar na primeira linha o maior valor negativo, fazendo com que a convergência da solução seja mais eficiente. O algoritmo deve copiar o tableu alocado na GPU para a memória comum antes desta etapa. O Quadro 24 contém o código que executa esse primeiro passo. As linha 2 e 3 criam variáveis locais que são usadas para verificar o menor valor negativo de uma lista de valores e identificar o menor índice. As linhas 4 a 10 contem o código que itera na lista fazendo a verificação do índice do menor valor.

Quadro 24 – Simplex: primeiro passo

```
1   int findIndiceK(double *SimplexTableau, int COLUNA) {
2     double menorValorNegativo = 0;
3     int indiceMenor = -1;
4     for(int i = 1; i < COLUNA; i++) {
5       double vAux = SimplexTableau[i];
6       if(menorValorNegativo > vAux) {
7         indiceMenor = i;
8         menorValorNegativo = vAux;
9       }
10    }
11    return indiceMenor;
12  }
```

Após encontrar qual a coluna de menor valor, deve ser feita a divisão do valor da primeira coluna com o corresponde da mesma linha na coluna encontrada no passo anterior. O Quadro 25 contém o código que executa esse segundo passo. A linha 4 faz a verificação de qual índice será utilizado no processamento, a linha7 faz a verificação se o valor é igual a zero e atribui um valor padrão neste caso, evitando com que o algoritmo tome valores inválidos em uma divisão por zero.

Quadro 25 – Simplex: segundo passo

```
1   __global__
2   void Kernel1(double *SimplexTableau, double *teta, double *Columnk, int k,
3                int COLUNA) {
4     int idx = blockDim.x * blockIdx.x + threadIdx.x; {
5       double w = SimplexTableau[COLUNA *idx + k];
6       Columnk[idx] = w;
7       if(w == 0.0) {
8          teta[idx] = 0.0;
9       }
10      else {
11         teta[idx] = SimplexTableau[COLUNA * idx] / w ;
12      }
13    }
14  }
```

No Quadro 26 está o código que verifica qual índice tem o menor valor. As linhas 2 e 3 tem declarações de variáveis locais usadas para verificar o índice do item de menor valor. As linhas 5 a 11 iteram em um `array` verificando qual índice do item de menor valor.

Quadro 26 – Simplex: terceiro passo

```
1   int findIndiceR(double *columk, int LINHA) {
2     double menorValor = (double)999999999999;//INT_MAX;
3     int indiceMenor = -1;
4
5     for(int i = 1; i < LINHA; i++) {
6       double vAux = columk[i];
7       if(menorValor > vAux && vAux > 0.0) {
8         indiceMenor = i;
9         menorValor = vAux;
10      }
11    }
12    return indiceMenor;
13  }
14
```

O Quadro 27 contém o código que atualiza a linha encontrada no passo 3.

Quadro 27 – Simplex: quarto passo

```
1   __global__
2   void Kernel2(double *SimplexTableau, double * Columnk, int k, int r, int
3   COLUNA) {
4     int idx = blockDim.x * blockIdx.x + threadIdx.x;
5     SimplexTableau[r * COLUNA + idx] =
6     SimplexTableau[r * COLUNA + idx] /(Columnk[r] );
7   }
```

O Quadro 28 contém o código que atualiza o tableau.

Quadro 28 – Simplex: quinto passo

```
1   __global__
2   void Kernel3(double *SimplexTableau, double *Columnk, int k, int r, int
3   COLUNA) {
4     int idx =  threadIdx.x;
5     int jdx =  blockIdx.x;
6     if (jdx == r)
7       return;
8     SimplexTableau[jdx * COLUNA + idx] = SimplexTableau[jdx * COLUNA + idx] –
9                                       Columnk[jdx]*SimplexTableau[r*COLUNA+idx];
10  }
```

O Quadro 29 contém o código que atualiza a coluna da variável de saída.

Quadro 29 – Simplex: sexto passo

```
1   __global__
2   void Kernel4(double *SimplexTableau, double * Columnk, int k, int r, int
3   COLUNA) {
4     int jdx =  threadIdx.x;
5     if(Columnk[r] != 0.0) {
6       SimplexTableau[jdx * COLUNA +  k] = -Columnk[jdx] /  ( Columnk[r] );
7     }
8
9     if (jdx == r) {
10      if(Columnk[r] != 0.0) {
11        SimplexTableau[jdx * COLUNA + k] = 1.0 / (Columnk[r]);
12      }
13    }
14  }
```

O Quadro 30 contém o código de verificação de parada do algoritmo.

Quadro 30 – Simplex: sétimo passo

```
1   bool stop(double *SimplexTableau, int COLUNA) {
2     for(int i = 1; i < COLUNA; i++) {
3       double vAux = SimplexTableau[i];
4       if(vAux < 0.0)
5          return false;
6       }
7       return true;
8     }
```

O Quadro 31 contém o fluxo básico da execução do algoritmo, sendo que algumas instruções foram omitidas, mostrando apenas o código mais relevante.

Quadro 31 – Fluxo principal

```
1   while(true) {
2     k = findIndiceK(SimplexTableau, COLUNA);
3     Kernel1<<<1,LINHA>>>(dev_tableau, dev_theta, dev_columK, k, COLUNA);
4     cudaMemcpy(teta, dev_theta, sizeColumn, cudaMemcpyDeviceToHost);
5     r = findIndiceR(teta, LINHA);
6     if(r == -1) {
7       printf("Problema Unbounded");
8       break;
9     }
10    Kernel2<<<COLUNA, 1>>>(dev_tableau, dev_columK, k, r, COLUNA);
11    cudaMemcpy(Columnk, dev_columK, sizeColumn, cudaMemcpyDeviceToHost);
12
13    Kernel3<<<LINHA, COLUNA >>>(dev_tableau, dev_columK, k, r, COLUNA);
14    cudaMemcpy(Columnk, dev_columK, sizeColumn, cudaMemcpyDeviceToHost);
15
16    Kernel4<<<1,LINHA>>>(dev_tableau, dev_columK, k, r, COLUNA);
17    cudaMemcpy(SimplexTableau,dev_tableau, sizeTableau,
18  cudaMemcpyDeviceToHost);
19   if(stop(SimplexTableau, COLUNA)) {
20      printSolution(SimplexTableau);
21      break;
22    }
23    if(iteration > LINHA ) {
24      printf("Não foi possível encontrar solucao");
25      break;
26    }
27     iteration++;
28  }
```

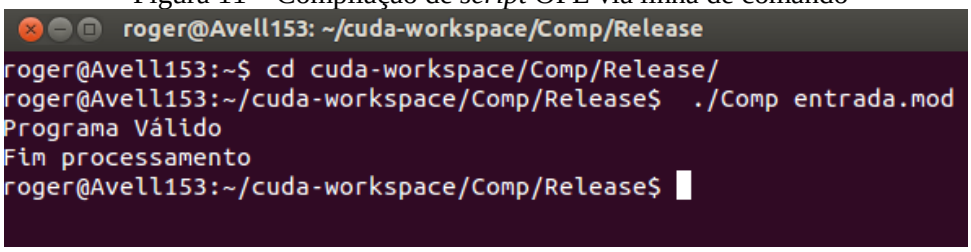
Na linha 2 é feita a chamada para a função `findIndiceK` e o resultado dessa função serve de parâmetro de entrada para a função `Kernel1`. Na linha 5 é chamada a função `findIndiceR` que retorna a coluna do maior valor negativo. A linha 6 verifica se a solução

permanece viável. São feitas chamadas às funções `Kernel2`, `Kernel3` e `Kernel4` para fazer as operações sobre o tableu. Na linha 19 é verificado se a rotina encontrou o resultado. Em caso positivo, o resultado será impresso no console.

3.3.5 Operacionalidade da implementação

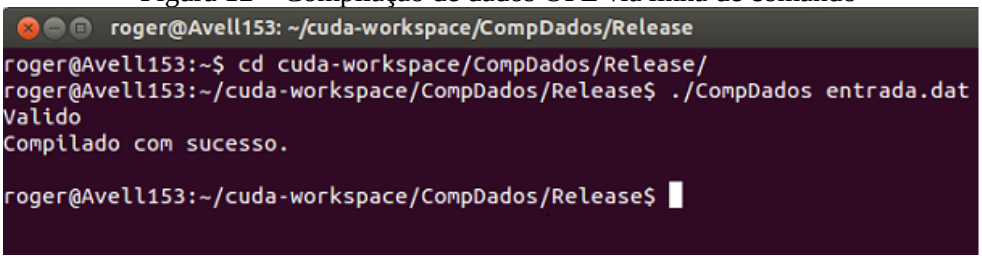
A ferramenta desenvolvida é composta por aplicativos C++ com interface via linha de comando. Para a compilação de *script* deve-se abrir um console e informar o nome da ferramenta (`comp`) e o arquivo a ser compilado. O resultado da execução será exibido no console, podendo ser: mensagem informando sucesso na compilação ou mensagem mostrando o erro ocorrido. Para *scripts* compilados com sucesso é gerado um arquivo com nome `Cplex.cpp` dentro da pasta `TesteCuda/src/`. A pasta `TesteCuda` deve estar no mesmo nível da pasta da ferramenta. A Figura 11 mostra a execução da ferramenta via linha de comando.

Figura 11 – Compilação de *script* OPL via linha de comando



Para a compilação de arquivos de dados deve-se abrir um console e informar o nome da ferramenta (`compdados`) e o arquivo a ser compilado. O resultado da execução será exibido no console, podendo ser: mensagem informando sucesso na compilação ou mensagem mostrando o erro ocorrido. Para arquivos de dados compilados com sucesso é gerado um arquivo de nome `Cplex.h` dentro da pasta `TesteCuda/src/`. A Figura 12 traz a execução da ferramenta via linha de comando.

Figura 12 – Compilação de dados OPL via linha de comando



Tanto a compilação de *scripts* quanto a compilação de arquivos de dados pode também ser executada dentro do ambiente de desenvolvimento Nsight Eclipse, obtendo a saída no console do próprio ambiente. Na Figura 13 e na Figura 14 são mostradas as compilações do *script* `knapsack.mod` e do arquivo de dados `knapsack.dat`, utilizando a IDE Nsight Eclipse.

Figura 13 – Compilação de *script* OPL a partir do Nsight Eclipse

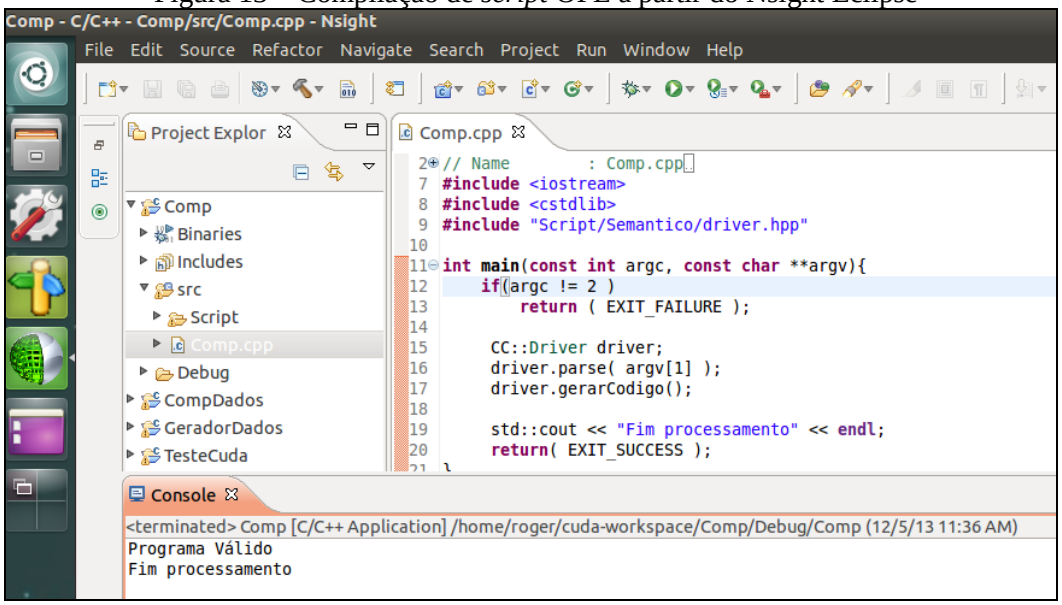
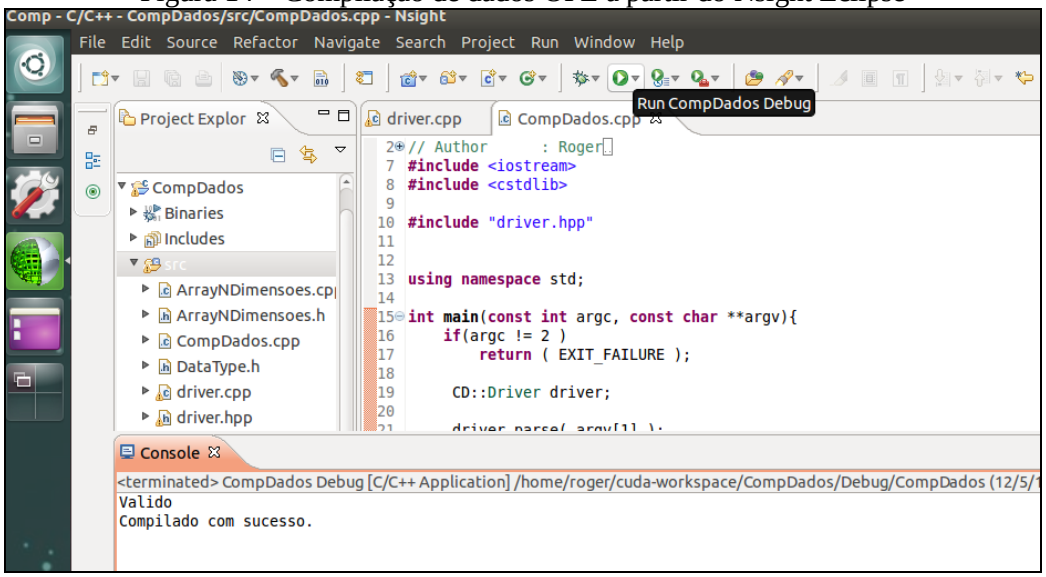


Figura 14 – Compilação de dados OPL a partir do Nsight Eclipse

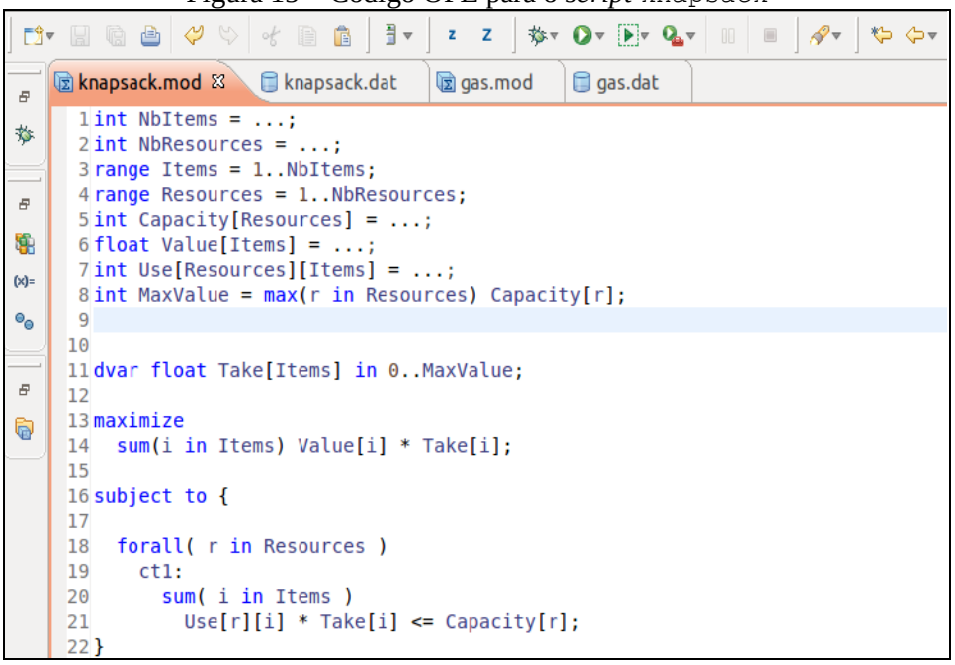


Para testar a ferramenta foram utilizados alguns *scripts* disponibilizados com a ferramenta IBM ILOG CPLEX. A fim realizar testes com uma massa de dados mais expressiva do que as originais, foram gerados dados aleatórios para preencher os valores de entrada do arquivo de dados.

O primeiro *script* testado foi o `knapsack`. É uma versão do problema da mochila, que consiste na escolha de um subconjunto de itens com base no valor do item. Devem ser escolhidos primeiro os itens de maior valor, fazendo com que o montante de itens dentro da mochila seja maximizado. Na Figura 15 é possível observar o código desse *script*. Observa-se que a quantidade escolhida de cada item tem uma restrição de capacidade máxima. O resultado é a quantidade de cada item que deve ser escolhida, sendo que a resposta será armazenada no vetor `Take` (linha 11). A função objetivo deste *script* é maximizar o somatório

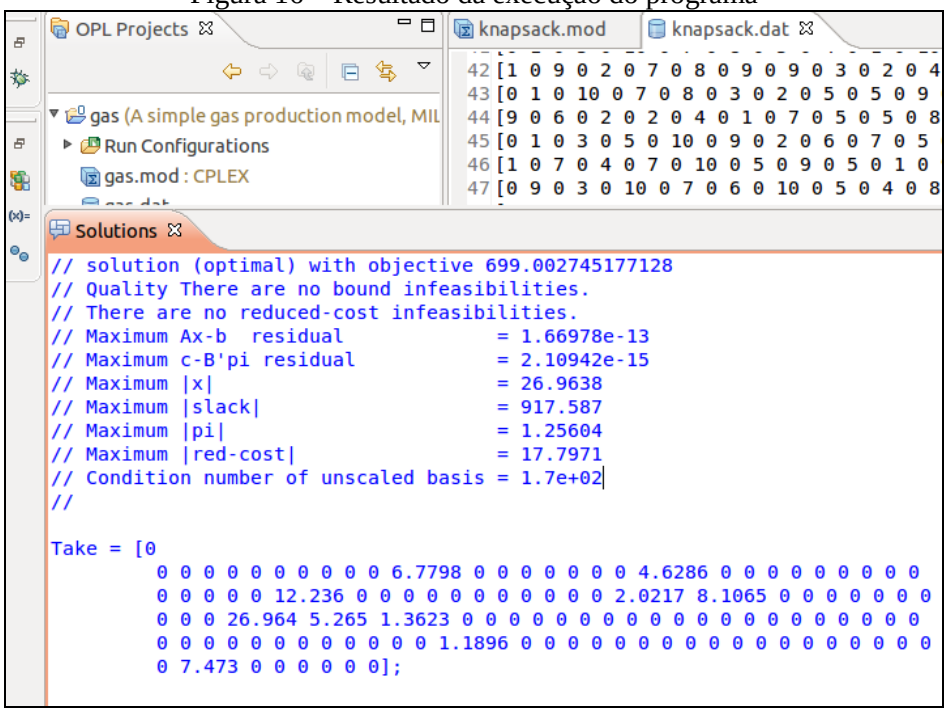
de quantos itens foram escolhidos de cada tipo, multiplicado pelo valor do objetivo, que fica armazenado no vetor `Value` (linhas 13 e 14). A restrição (`ct1`) garante que só podem ser escolhidos itens (`Use`) que não ultrapassem a capacidade (`Capacity`) de recursos disponíveis (linhas 16 a 21).

Figura 15 – Código OPL para o *script* `knapsack`



A Figura 16 apresenta o resultado da execução do *script* usando a ferramenta IBM ILOG CPLEX.

Figura 16 – Resultado da execução do programa



A partir do *script* da Figura 15 foi gerado código correspondente em C++ usando a

ferramenta desenvolvida. O código do *script*, convertido para código C++, é mostrado no Quadro 32. O resultado da execução do programa de teste é exibido no Quadro 33.

Quadro 32 – Código gerado pelo protótipo

```
#include <vector>
#include "Cplex.h"
#include "Solver/SimplexSolver.h"
#include "Solver/Restricao.h"
#include "Solver/Objetivo.h"
using namespace std;
using namespace CC;
vector<Restricao> restricoes;
Objetivo obj;
SimplexSolver solver;
int sizeArray;
void processar() {
int codigoRetorno = solver.Solve();
if(codigoRetorno == 0)
   cout << "Ok";
else
   cout << "Erro ao procesar, codigo do erro" << codigoRetorno;
}
struct Range {
int minValue, maxValue;
Range(int min, int max) {
  minValue = min;
  maxValue = max;
}
int size() {
       return (maxValue - minValue) + 1;
}
};
int main(const int argc, const char **argv) {
int NbItems NbItemsVal;
int NbResources NbResourcesVal;
Range Items(1, NbItems);
Range Resources(1, NbResources);
int Capacity CapacityVal;
double Value ValueVal;
int Use UseVal;
int MaxValue = NULL;
load();
//Objetivo
sizeArray = sizeof(Value) / sizeof(Value[0]);
for(int i = 0; i < sizeArray; i++) {
   obj.valores.push_back(Value[i]);
}
//Restricoes
for(int r= 0;r < Resources.size(); r++) {
Restricao _restricao;
_restricao.id = "ct1";
for(int i = 0; i < Items.size(); i++) {
   _restricao.pesos.push_back(Use[r][i]);
}
//Limite
_restricao.limite = Capacity[r];
restricoes.push_back(_restricao);
}
solver.MontarTableu(restricoes, obj);
processar();
}
```

Quadro 33 – Resultado da execução do programa

```
restricoes =70 objetivos =120
Alocaçao ok


Solução  = 699.002745
Bases
x11 = 6.779755
x19 = 4.628553
x34 = 12.235507
x46 = 2.021739
x47 = 8.106491
x58 = 26.963768
x59 = 5.265008
x60 = 1.362319
x93 = 1.189646
x113 = 7.472989

Ok
```

Outros dois *scritps* do pacote de exemplos disponíveis junto com a ferramenta IBM ILOG CPLEX também foram executados com sucessso, sendo eles: `gas` e `production`.

3.4 RESULTADOS E DISCUSSÃO

Não foi encontrado no decorrer deste trabalho nenhuma iniciativa de processamento de *scripts* OPL, com excessão da própria IBM ILOG CPLEX. A Tabela 1 mostra um comparativo dos resultados obtidos para as variáveis decisórias no caso de teste proposto anteriormente na IBM ILOG CPLEX e na ferramenta desenvolvida. Na IBM ILOG CPLEX os valores tem 4 casas decimais, enquanto na ferramenta desenvolvida foram consideradas as 6 primeiras casas decimais.

Tabela 1 – Comparativo dos resultados

variável	IBM ILOG CPLEX	ferramenta desenvolvida
Total	699.002745177	699.002745
x11	6.7798	6.779755
x19	4.6286	4.628553
x34	12.236	12.235507
x46	2.0217	2.021739
x47	8.1065	8.106491
x58	26.964	26.963768
x59	5.265	5.265008
x60	1.3623	1.362319
x93	1.1896	1.189646
x113	7.473	7.472989

No Quadro 34 é apresentada uma comparação entre a ferramenta desenvolvida e os trabalhos correlatos apresentados.

Quadro 34 – Comparativo dos trabalhos correlatos com a ferramenta desenvolvida

característica / ferramenta	Gesser (2007)	Lalami, Boyer, El-Baz (2011)	Moreira, Meneses (2012)	ferramenta proposta
linguagem fonte	Java	-	-	OPL
linguagem objeto	C++	C++	C++	C++
plataforma de execução	Palm OS	CPU + GPU	CPU + GPU	CPU + GPU
algoritmo de otimização combinatória	não se aplica	Simplex	Tabu Search	Simplex

Neste trabalho foi apresentado o desenvolvimento de uma ferramenta que faz a compilação de *scripts* e de arquivos de dados OPL com geração de código C++. Os resultados obtidos alcançaram os objetivos. No entanto, devido à quantidade de construções sintáticas dos *scripts* OPL, à compreensão parcial da semântica e ao fato de que algumas construções sintáticas utilizam outros algoritmos de resolução diferentes do Simplex, não foi possível tratar totalmente a linguagem com o compilador desenvolvido. A regra utilizada para delimitar o que seria tratado foi garantir que no mínimo a semântica e a geração de código das construções utilizadas nos estudos de caso desenvolvidos para testar o compilador fossem respeitadas. Já a gramática dos arquivos de dados OPL é mais simples, logo foram tratados todos recursos que são suportados, exceto o recurso de utilização de dados oriundos de banco de dados ou arquivos Excel.

4 CONCLUSÕES

Com a pesquisa sobre otimização combinatória, verificou-se a necessidade de uma ferramenta de apoio à resolução de problemas desta natureza. Observou-se que a área de otimização combinatória usando programação OPL já possui implementações que funcionam e com resultados excelentes. Entretanto, verificou-se que não utilizam o paralelismo em placas de vídeo. Os algoritmos que resolvem estes problemas, como Simplex, podem ser desenvolvidos usando paralelismo, como visto nos trabalhos correlatos. Assim sendo, foi desenvolvida uma ferramenta para converter *scripts* em OPL para código correspondente em linguagem C/C++, com algoritmos paralelos em GPU.

O objetivo principal do trabalho foi alcançado com sucesso. O compilador desenvolvido mostrou-se capaz de detectar os erros léxicos, sintáticos e semânticos nos programas testados. No entanto, não foi possível desenvolver toda a análise semântica da gramática, mas uma parte importante foi implementada. A execução do programa gerado pela ferramenta apresentou precisão aceitável nos resultados comparados à ferramenta IBM ILOG CPLEX, com poucas divergências de arredondamento de valores.

Observa-se que não foi possível realizar uma comparação de tempo de execução com a ferramenta IBM ILOG CPLEX devido à restrição de tamanho do problema modelado na versão gratuita (no máximo 500 restrições). Porém, conforme os trabalhos correlatos, o ganho de tempo de execução de programas em GPU aumenta conforme o tamanho do problema e os resultados se mantiveram com precisão aceitável com um número grande de variáveis.

A exemplo dos trabalhos correlatos, existem diversas pesquisas na área de utilização de processamento gráfico para resolução de problemas de programação linear. Logo, a execução de *scripts* OPL na plataforma GPU com a ferramenta desenvolvida provê uma forma de executar os programas já criados. Além disso, a elaboração e a manutenção desses *scripts* em OPL é mais simples do que a respectiva implementação em C++ com GPU, visto que os problemas em OPL podem ser modelados por pessoas da área de negócio ou da matemática que não estejam familiarizadas com C++ para GPU.

4.1 EXTENSÕES

Sugere-se como extensões a este trabalho:

a) concluir o analisador semântico e o gerador de código, dando suporte a todas as construções da gramática;
b) implementar outros algoritmos para resolver problemas de programação linear;
c) suportar entrada de dados de banco de dados e arquivos do Excel.

d) portar o algoritmo Simplex para outras plataformas de placas de vídeo;
e) automatizar a compilação e a execução para que o usuário tenha que interagir menos com a ferramenta.

REFERÊNCIAS BIBLIOGRÁFICAS

AHO, Alfred V. et al. **Compiladores:** princípios, técnicas e ferramentas. 2. ed. Tradução Daniel Vieira. São Paulo: Pearson Addison-Wesley, 2008.

BARR, Richard S. et al. Designing and reporting on computational experiments with heuristic methods. **Journal of Heuristics**, [S.l.], v. 1, p. 9-32, 1995. Disponível em: <http://www2.research.att.com/~mgcr/doc/guidelines.pdf>. Acesso em: 27 maio 2013.

DANTZIG, George**. Linear programming and extensions**. 11th ed. Princeton: Princeton University Press, 1998.

GESSER, Júlio V. **Compilador Java 5.0 para gerar código C++ para plataforma Palm OS**. 2007. 84 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) - Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau. Disponível em: <http://campeche.inf.furb.br/tccs/2007-I/2007-1juliovilmargesservf.pdf>. Acesso em: 28 mar. 2013.

GRUNE, Dick. et al. **Projeto moderno de compiladores**: implementação e aplicações. Tradução Vandenberg D. de Souza. Rio de Janeiro: Campus, 2001.

IBM SOFTWARE. **Getting started with CPLEX.** [S.l.], [2007]. Disponível em: <http://pic.dhe.ibm.com/infocenter/cosinfoc/v12r2/index.jsp?topic=%2Filog.odms.cplex.help%2FContent%2FOptimization%2FDocumentation%2FCPLEX%2F_pubskel%2FCPLEX23.html>. Acesso em: 25 abr. 2013.

_____. **IBM ILOG CPLEX optimization studio**. [S.l.], [2011a]. Disponível em: <http://www-142.ibm.com/software/products/br/pt/ibmilogcpleoptistud>. Acesso em: 01 abr. 2013.

_____. **IBM ILOG CPLEX**: IDE and OPL. [S.l.], [2011b]. Disponível em: <http://pic.dhe.ibm.com/infocenter/cosinfoc/v12r3/index.jsp?topic=%2Filog.odms.ide.help%2FContent%2FOptimization%2FDocumentation%2FOptimization_Studio%2F_pubskel%2Fps_opl_Language1149.html>. Acesso em: 14 dez. 2013.

LALAMI, Mohamed E.; BOYER, Vincent; EL-BAZ, Didier. Efficient implementation of the Simplex method on a CPU-GPU system. In: IEEE INTERNATIONAL SYMPOSIUM ON PARALLEL AND DISTRIBUTED PROCESSING WORKSHOPS AND PHD FORUM, 11., 2011, Washington. **Proceeding...** Washington: IEEE Computer Society, 2011. p. 1999-2006. Disponível em: <http://homepages.laas.fr/elbaz/PCO11.pdf>. Acesso em 02: mar. 2013.

LOUDEN, Kenneth. **Compiladores**: princípios e práticas. Tradução Flávio Soares Corrêa da Silva. São Paulo: Pioneira Thomson Learning, 2004.

MOREIRA, Eduardo B. G.; MENESES, Cláudio N. Algoritmos paralelos em GPUs para problemas de programação quadrática binária irrestrita. In: SIMPÓSIO BRASILEIRO DE PESQUISA OPERACIONAL, 44., 2012, Rio de Janeiro. **Anais eletrônicos...** Rio de Janeiro: FGV, 2012. Não paginado. Disponível em: <http://www2.claiosbpo2012.iltc.br/pdf/102151.pdf >. Acesso em: 05 mar. 2013

NAZARETH, John L. **An optimization primer**. Nova York: Springer-Verlag, 2004.

NVIDIA CORPORATION. **CUDA C programming guide**. [S.l.], [2012a]. Disponível em: <http://docs.nvidia.com/cuda/cuda-c-programming-guide/>. Acesso em: 04 abr. 2013.

_____. **O que é computação com GPU?** [S.l.], [2012b]. Disponível em: <http://www.nvidia.com.br/object/what-is-gpu-computing-br.html>. Acesso em: 01 abr. 2013.

OCHI, Luiz S.; DRUMMOND, Lúcia M. A.; SILVA, Mozar B. **Metaheurística GRASP/VNS para a solução de problemas de otimização combinatória**. [Niterói], 2000. Disponível em: <http://www2.ic.uff.br/~satoru/conteudo/artigos/sbpo-Mozar-2000.pdf>. Acesso em: 27 maio 2013.

PIWAKOWSKI, Konrad. Applying Tabu search to determine new Ramsey graphs. **The Electronic Journal of Combinatorics**, [S.l.], v. 3, n. 1, p. 1-4, 1996. Disponível em: <http://www.combinatorics.org/ojs/index.php/eljc/article/view/v3i1r6/pdf>. Acesso em: 01 maio 2013.

PRICE, Ana M. A.; TOSCANI, Simão S. **Implementação de linguagens de programação:** compiladores. Porto Alegre: Sagra Luzzarto, 2001.

TAVARES, Luíz V.; CORREIA, Francisco N. **Optimização linear e não linear**: conceitos, métodos e algoritmos. Lisboa: Fundação Calouste Gulbenkian, 1999.

WATT, David A.; BROWN, Deryck F. **Programming language processors in Java**. Harlow: Person Education Limited, 2000.

APÊNDICE A – Especificação léxica de *script* OPL

No Quadro 35 está a especificação léxica para *scripts* OPL.

Quadro 35 – Especificação léxica de *script* OPL

```
%{
#include <string>
#include <cstdlib>
#include <iostream>
using namespace std;
#include "scanner.hpp"
int lineNumber = 1;
typedef CC::Parser::token token;
#define STOKEN( x ) ( new std::string( x ) )
#define yyterminate() return( token::END )
#define YY_NO_UNISTD_H
%}

%option debug
%option nodefault
%option yyclass="Scanner"
%option noyywrap
%option c++
%%

[ \t\r]      ;
[\n]         {++lineNumber;}
";"          {return( token::SEMICOLON ); }
","          {return (token::COMMA);}
":"          {return (token::COLON); }
"if"         {return( token::IF );  }
"minimize"   {return( token::MINIMIZE );}
"maximize"   {return( token::MAXIMIZE ); }
"include"    {return( token::INCLUDE );}
"using"      {return ( token::USING );}
"error"      {return (token::ERROR) ; }
"execute"    {return (token::EXECUTE);}
"main"       {return (token::MAIN);}
"constraints"{return (token::CONSTRAINTS); }
"assert"     {return (token::ASSERT); }
"dvar"       {return (token::DVAR);}
"dexpr"      {return (token::DEXPR);}
"="          {return (token::ATRIB);}
"subject"    {return (token::SUBJECT);}
"to"         {return (token::TO);}
"in"         {return (token::IN);}
"tuple"      {return (token::TUPLE);}
"and"        {return (token::AND );}
"or"         {return (token::OR );}
"prod"       {return (token::PROD );}
"max"        {return (token::MAX );}
"min"        {return (token::MIN );}
"sum"        {return (token::SUM );}
"symdiff"    {return (token::SYMDIFF );}
"diff"       {return (token::DIFF );}
"union"      {return (token::UNION );}
"inter"      {return (token::INTER );}
"not"        {return (token::NOT );}
"mod"        {return (token::MOD );}
"div"        {return (token::DIV );}
"maxint"     {return (token::MAXINT );}
"infinity"   {return (token::INFINITY );}
"."          {return (token::POINT );}
```

```
".."          {return (token::TWOPOINTS );}
"..."         {return (token::THREEPOINTS );}
"["           {return (token::COLCHETESESQ );}
"]"           {return (token::COLCHETESDIR );}
"("           {return (token::PARENTESESESQ );}
")"           {return (token::PARENTESESDIR );}
"{"           {return (token::CHAVEE );}
"}"           {return (token::CHAVED );}
"piecewise"   {return (token::PIECEWISE );}
"else"        {return (token::ELSE );}
"forall"      {return (token::FORALL );}
"reversed"    {return (token::REVERSED );}
"sorted"      {return (token::SORTED );}
"ordered"     {return (token::ORDERED );}
"setof"       {return (token::SETOF );}
"constraint" {return (token::CONSTRAINT );}
"range"       {return (token::RANGE );}
"float+"      {return (token::FLOATPLUS );}
"int+"        {return (token::INTPLUS );}
"boolean"     {return (token::BOOLEAN );}
"float"       {return (token::FLOAT);}
"int"         {return (token::INT );}
"key"         {return (token::KEY );}
"with"        {return (token::WITH );}
"struct"      {return (token::STRUCT );}
"all"         {return (token::ALL );}
"=="          {return (token::EQUALS);}
"string"      {return (token::STRING);}
"^"           {return (token::XOR );}
"&&"          {return (token::AND );}
"||"          {return (token::OR_EXPRESSION );}
"=>"          {return (token::IMPLY );}
"-"           {return (token::MINUS );}
"+"           {return (token::PLUS );}
"*"           {return (token::MULTIPLY );}
"/"           {return (token::DIVIDE);}
"%"           {return (token::PERCENT);}
"!"           {return (token::NEGATION);}
"!="          {return (token::NOT_EQUALS);}
"<="          {return (token::LE );}
"<"           {return (token::LT );}
">="          {return (token::GE );}
">"           {return (token::GT );}
"?"           {return (token::QUESTION_MARK);}
"->"          {return (token::POINTER);}
"|"           {return (token::OR_BINARY); }
[0-9]+        {yylval->sval = STOKEN(yytext);      return (token::INTEGERLITERAL);}
 [0-9]*"."[0-9]+ {yylval->sval = STOKEN(yytext); return (token::FLOATLITERAL);}
"true"        {yylval->sval = STOKEN(yytext);     return (token::BOOLEANLITERAL);}
"false"       {yylval->sval = STOKEN(yytext);     return (token::BOOLEANLITERAL);}
[\"][^\"]*[\"] {yylval->sval = STOKEN(yytext);   return (token::STRINGLITERAL);}
[_a-zA-Z][_a-zA-Z0-9]* {yylval->sval = STOKEN( yytext ); return( token::ID );}
"/*"          {int c;
               while((c = yyinput()) != 0) {
                 if(c == '\n')
                   ++lineNumber;
                 else if(c == '*') {
                   if((c = yyinput()) == '/')
                     break;
                   else
                     unput(c);
                   }
                 }
               }
"//".*        ;
.             ;
%%
```

APÊNDICE B – Especificação sintática de *script* OPL

No Quadro 36 está a especificação usada na criação do analisador sintático do compilador de *scripts* OPL.

Quadro 36 – Especificação sintática para *scripts* OPL usando Bison

```
%skeleton "lalr1.cc"
%require  "2.7"
%locations
%error-verbose

%debug

%code requires{
   namespace CC {
      class Driver;
      class Scanner;
   }
}

/* set the parser's class identifier */
%define "parser_class_name" "Parser"

/* namespace to enclose parser in */
%name-prefix="CC"

%lex-param   { Scanner  &scanner  }
%parse-param { Scanner  &scanner  }

%lex-param   { Driver  &driver  }
%parse-param { Driver  &driver  }

%code{
   #include <iostream>
   #include <cstdlib>
   #include <fstream>
   #include <stdio.h>
   #include "../Semantico/DataType.h"
   #include "../Semantico/SemanticAction.h"

   extern int lineNumber;
   /* include for all driver functions */
   #include "../Semantico/driver.hpp"
   #define YYERROR_VERBOSE 1

   /* this is silly, but I can't figure out a way around */
   static int yylex(CC::Parser::semantic_type *yylval,
CC::Parser::location_type* loc,
                    CC::Scanner  &scanner,
                    CC::Driver   &driver);


}

/* token types */
%union {
   int  iVal;
   float fVal;
   bool bVal;
   std::string *sval;
}
```

```
%token END       0     "end of file"
%token NEWLINE
%token SEMICOLON
%token MAXIMIZE
%token MINIMIZE
%token IF
%token INCLUDE
%token USING
%token TEMPLATE
%token ERROR
%token EXECUTE
%token MAIN
%token CONSTRAINTS
%token ASSERT
%token DVAR
%token DEXPR
%token ATRIB
%token SUBJECT
%token TO
%token IN
%token TUPLE
%token STRUCT
%token WITH
%token KEY
%token INT
%token FLOAT
%token BOOLEAN
%token INTPLUS
%token FLOATPLUS
%token RANGE
%token CONSTRAINT
%token SETOF
%token ORDERED
%token SORTED
%token REVERSED
%token FORALL
%token ELSE
%token CHAVED
%token CHAVEE
%token PIECEWISE
%token PARENTESESDIR
%token PARENTESESESQ
%token COMMA
%token COLCHETESDIR
%token COLCHETESESQ
%token POINT
%token TWOPOINTS
%token THREEPOINTS
%token INFINITY
%token MAXINT
%token DIV
%token MOD
%token NOT
%token INTER
%token UNION
%token DIFF
%token SYMDIFF
%token SUM
%token MIN
%token MAX
%token PROD
%token OR
%token AND
%token ALL
%token EQUALS
%token STRING
%token XOR
%token OR_EXPRESSION
```

```
%token IMPLY
%token MINUS
%token PLUS
%token NEGATION
%token NOT_EQUALS
%token LE
%token LT
%token GE
%token GT
%token MULTIPLY
%token DIVIDE
%token PERCENT
%token COLON
%token QUESTION_MARK
%token POINTER
%token OR_BINARY

%token <sval> INTEGERLITERAL
%token <sval> FLOATLITERAL
%token <sval> BOOLEANLITERAL
%token <sval> STRINGLITERAL
%token <sval> ID

/* destructor rule for <sval> objects */
%destructor { if ($$)  { delete ($$); ($$) = NULL; } } <sval>


%%
CompilationUnit: TopLevelDeclarations_opt
                       ;
TopLevelDeclarations_opt: /* empty */
                        | TopLevelDeclarations
                        ;

TopLevelDeclarations: TopLevelDeclaration
                    | TopLevelDeclarations TopLevelDeclaration
                    ;

TopLevelDeclaration: INCLUDE STRINGLITERAL SEMICOLON {
driver.addStringLiteral(*$2); driver.action(INCLUDE); }
                                | InModelDeclaration
                    | USING ID SEMICOLON { driver.usarAlgoritmo(*$2);
driver.action(USING);}
                    | Model
                    | ERROR SEMICOLON
                                ;

Scripting: EXECUTE Id_opt STRINGLITERAL { driver.addStringLiteral(*$3);
driver.action(EXECUTE); }
         | MAIN STRINGLITERAL { driver.addStringLiteral(*$2);
driver.action(MAIN); }
         ;

Model: TEMPLATE ID CHAVEE InModelDeclarations_opt CHAVED
      ;

Id_opt: /* empty */
       | ID
       ;

InModelDeclarations_opt: /* empty */
                       | InModelDeclarations
                       ;

InModelDeclarations: InModelDeclaration
                   | InModelDeclarations InModelDeclaration
                   ;
```

```
 InModelDeclaration: LocalVar SEMICOLON
                   | TypeDeclaration {driver.action(32); }
                   | Objective SEMICOLON {driver.action(OBJECTIVE); }
                   | CONSTRAINTS CHAVEE Constraints_opt CHAVED {
driver.action(CONSTRAINTS); }
                   | SUBJECT TO CHAVEE Constraints_opt CHAVED  {
driver.action(SUBJECTTO); }
                   | SEMICOLON {driver.action(34); }
                   | ASSERT Constraint {driver.action(35); }
                   | Scripting  {driver.action(36); }
                   ;

LocalVar: Type VariableDeclarator { driver.action(VAR); }
         | DVAR Type VariableDeclarator DecisionVarRange_opt {
driver.action(DVAR1); }
         | DVAR Type VariableDeclarator ATRIB Expression { driver.action(DVAR2);
}
         | DEXPR Type VariableDeclarator ATRIB Expression {
driver.action(DEXPR);}
         | Type VariableDeclarator ATRIB Expression { driver.action(VAR1);}
         | Type VariableDeclarator ATRIB THREEPOINTS {
driver.action(THREEPOINTS); }
         ;

DecisionVarRange_opt: /* empty */ {driver.action(37); }
                    | DecisionVarRange {driver.action(38); }
                    ;

DecisionVarRange: IN Expression {driver.action(39); }
                        ;

VariableDeclarator: ID Dimensions_opt References_opt { driver.addId(*$1); }
{driver.action(40); }
                          ;

Dimensions_opt: /* empty */
               | Dimensions {driver.action(41); }
               | COLCHETESESQ ManyExpressions COLCHETESDIR {driver.action(42); }
               | COLCHETESESQ SimpleQualifiers COLCHETESDIR {driver.action(43);
}
               ;

Dimensions: Dimension {driver.action(44); }
           | Dimensions Dimension {driver.action(45); }
           ;

Dimension: COLCHETESESQ Expression COLCHETESDIR { driver.action(DIMENSION1);}
          | COLCHETESESQ SimpleQualifier COLCHETESDIR {
driver.action(DIMENSION2);}
          ;

References_opt: /* empty */
              | WITH References {driver.action(46); }
              ;

References: Reference {driver.action(47); }
           | References COMMA Reference {driver.action(48); }
           ;

Reference: Ids IN ID {driver.action(49); }
              ;

Ids: ID {driver.action(50); }
    | Ids COMMA ID {driver.action(51); }
    ;

TypeDeclaration: TUPLE ID CHAVEE TupleComponents CHAVED {driver.action(52); }
```

```
                | STRUCT ID CHAVEE TupleComponents CHAVED {driver.action(53); }
                ;

 TupleComponents: TupleComponent {driver.action(54); }
                | TupleComponents TupleComponent {driver.action(55); }
                ;

 TupleComponent: Type VariableDeclarator SEMICOLON {driver.action(56); }
               | KEY Type VariableDeclarator SEMICOLON {driver.action(57); }
               ;

Type: NonArrayType {driver.action(58); }
     | Type COLCHETESESQ Type COLCHETESDIR {driver.action(59); }
     | SetType {driver.action(60); }
     ;

NonArrayType: LiteralType {driver.action(61); }
             | ID { driver.addId(*$1); } {driver.action(62); }
             ;

LiteralType: INT { driver.empilhaTipoDado( INT ); }
            | FLOAT { driver.empilhaTipoDado( FLOAT); }
            | BOOLEAN { driver.empilhaTipoDado( BOOLEAN ); }
            | INTPLUS { driver.empilhaTipoDado( UINT ); }
            | FLOATPLUS { driver.empilhaTipoDado( UFLOAT ); }
            | STRING { driver.empilhaTipoDado( STRING ); }
            | RANGE { driver.empilhaTipoDado( RANGE ); }
            | RANGE FLOAT { driver.empilhaTipoDado( RANGE_FLOAT ); }
            | CONSTRAINT  { driver.empilhaTipoDado( CONSTRAINT ); }
            ;

SetType: SETOF PARENTESESESQ NonArrayType PARENTESESDIR {driver.action(63); }
        | CHAVEE NonArrayType CHAVED {driver.action(64); }
        | SetTypeModifier SETOF PARENTESESESQ NonArrayType PARENTESESDIR
{driver.action(65); }
        | SetTypeModifier CHAVEE NonArrayType CHAVED {driver.action(66); }
        ;

SetTypeModifier: ORDERED {driver.action(67); }
                | SORTED {driver.action(68); }
                | REVERSED {driver.action(69); }
                ;

Constraints_opt: /* empty */
                | Constraints {driver.action(70); }
                ;

Constraints: Constraint {driver.action(71); }
            | Constraints Constraint {driver.action(72); }
            ;

Constraint: Expression SEMICOLON {driver.action(73); }
           | ID COLON Constraint {driver.addId(*$1); driver.action(74); }
           | ArraySlotExpression COLON Constraint {driver.action(75); }
           | LocationExpression ATRIB Constraint {driver.action(76); }
           | IF PARENTESESESQ Expression PARENTESESDIR Constraint ELSE
Constraint {driver.action(77); }
           | IF PARENTESESESQ Expression PARENTESESDIR Constraint
{driver.action(78); }
           | FORALL {driver.action(FORALLCONSTRUCTOR);} PARENTESESESQ Qualifiers
PARENTESESDIR {driver.action(FORALLCONSTRAINT);} Constraint
{driver.action(FORALL);}
           | FORALL {driver.action(FORALLCONSTRUCTOR);} PARENTESESESQ Qualifiers
PARENTESESDIR {driver.action(FORALLCONSTRAINT);} ArraySlotExpression ATRIB
Constraint {driver.action(FORALL2);}
           | BooleanBlock {driver.action(79); }
           | SEMICOLON {driver.action(80); }
           | ERROR SEMICOLON
```

```
            | ERROR CHAVED
            ;

Objective:  MINIMIZE Expression { driver.action(MINIMIZE);}
          | MAXIMIZE Expression { driver.action(MAXIMIZE);}
          ;

PathExpression: LocationExpression {driver.action(81); }
              | ArraySlotExpression {driver.action(82); }
              | FunctionCall {driver.action(83); }
              ;

Expression: PARENTESESESQ Expression PARENTESESDIR
            | PathExpression {driver.action(84); }
            | LiteralExpression {driver.action(28); }
            | BinaryExpression {driver.action(BINARYEXPRESSION); }
            | UnaryExpression {driver.action(87); }
            | RangeExpression {driver.action(88); }
            | AggregateExpression {driver.action(89); }
            | CollectionExpression {driver.action(90); }
            | TupleExpression {driver.action(91); }
            | PiecewiseExpression {driver.action(92); }
            | Expression QUESTION_MARK Expression COLON Expression
{driver.action(93); }
            ;

LiteralExpression: INTEGERLITERAL  { driver.empilhaLiteral(*$1); }
                 | FLOATLITERAL       { driver.empilhaLiteral(*$1);}
                 | BOOLEANLITERAL { driver.empilhaLiteral(*$1); }
                 | STRINGLITERAL { driver.empilhaLiteral(*$1); }
                 | INFINITY {driver.action(94);}
                 | MAXINT {driver.action(95); }
                 ;

 BinaryExpression: Expression EQUALS Expression { driver.addOperadorBinario(1);}
                 | Expression NOT_EQUALS Expression {
driver.addOperadorBinario(2); }
                 | Expression LE Expression { driver.addOperadorBinario(3);}
                 | Expression LT Expression { driver.addOperadorBinario(4);}
                 | Expression GE Expression { driver.addOperadorBinario(5); }
                 | Expression GT Expression { driver.addOperadorBinario(6); }
                 | Expression PLUS Expression { driver.addOperadorBinario(7); }
                 | Expression MINUS Expression { driver.addOperadorBinario(8); }
                 | Expression MULTIPLY Expression {
driver.addOperadorBinario(9);}
                 | Expression DIVIDE Expression { driver.addOperadorBinario(10);
}
                 | Expression DIV Expression { driver.addOperadorBinario(11); }
                 | Expression PERCENT Expression {
driver.addOperadorBinario(12); }
                 | Expression MOD Expression { driver.addOperadorBinario(13); }
                 | Expression IN Expression { driver.addOperadorBinario(14); }
                 | Expression NOT IN Expression { driver.addOperadorBinario(15);
}
                 | Expression INTER Expression { driver.addOperadorBinario(16);
}
                 | Expression UNION Expression { driver.addOperadorBinario(17);
}
                 | Expression DIFF Expression { driver.addOperadorBinario(18); }
                 | Expression SYMDIFF Expression {
driver.addOperadorBinario(19); }
                 | Expression XOR Expression { driver.addOperadorBinario(20); }
                 | Expression AND Expression { driver.addOperadorBinario(21); }
                 | Expression OR_EXPRESSION Expression {
driver.addOperadorBinario(22); }
                 | Expression IMPLY Expression { driver.addOperadorBinario(23);
}
                ;
```

```
UnaryExpression: MINUS Expression {driver.action(116); }
                | NEGATION Expression {driver.action(117); }
                ;

LocationExpression: ID { driver.addId(*$1); driver.action(172);}
                    | PathExpression POINT ID {driver.action(118); }
                    | PARENTESESESQ AllExpression PARENTESESDIR
{driver.action(119); }
                    ;

ArraySlotExpression: PathExpression COLCHETESESQ Expression COLCHETESDIR {
driver.action(ARRAYSLOTEXPRESSION);}
                    | PathExpression COLCHETESESQ ManyExpressions COLCHETESDIR
{driver.action(120); }
                    ;

Locations: LocationExpression COMMA LocationExpression {driver.action(121); }
          | Locations COMMA LocationExpression {driver.action(122); }
          ;

Expressions: Expression
            | Expressions COMMA Expression {driver.action(123); }
            ;

ManyExpressions: Expression COMMA Expression {driver.action(124); }
                | ManyExpressions COMMA Expression {driver.action(125); }
                ;

Expressions_opt: /* empty */
                | Expressions
                ;

RangeExpression: Expression TWOPOINTS Expression {driver.action(126); }
                       ;

BooleanBlock: CHAVEE BlockExpressions CHAVED {driver.action(127); }
                     ;

BlockExpressions: BlockExpression {driver.action(128); }
                  | BlockExpressions BlockExpression {driver.action(129); }
                  ;

BlockExpression: Constraint {driver.action(130); }
                           ;

TupleExpression: LT Expressions GT {driver.action(131); }
                  | LT ERROR GT {driver.action(132); }
                  ;

FunctionCall: ID PARENTESESESQ Expressions_opt PARENTESESDIR
{driver.action(133); }
                   ;


AggregateExpression: SUM PARENTESESESQ Qualifiers PARENTESESDIR Expression {
driver.action(SUM);}
                     | MIN PARENTESESESQ Qualifiers PARENTESESDIR Expression
{driver.action(134); }
                     | MAX PARENTESESESQ Qualifiers PARENTESESDIR Expression
{driver.action(135); }
                     | PROD PARENTESESESQ Qualifiers PARENTESESDIR Expression
{driver.action(136); }
                     | OR PARENTESESESQ Qualifiers PARENTESESDIR Expression
{driver.action(137); }
                     | AND PARENTESESESQ Qualifiers PARENTESESDIR Expression
{driver.action(138); }
                     | UNION PARENTESESESQ Qualifiers PARENTESESDIR Expression
```

```
{driver.action(139); }
                    | INTER PARENTESESESQ Qualifiers PARENTESESDIR Expression
{driver.action(140); }
                    | AllExpression {driver.action(141); }
                    ;

AllExpression: ALL AllRange_opt PARENTESESESQ Qualifiers PARENTESESDIR
Expression {driver.action(142); }
                     ;

AllRange_opt: /* empty */ {driver.action(143); }
             | AllRange {driver.action(144); }
             ;

AllRange: COLCHETESESQ RangeExpression COLCHETESDIR {driver.action(145); }
         | COLCHETESESQ LocationExpression COLCHETESDIR {driver.action(146); }
         | COLCHETESESQ Expression TWOPOINTS MULTIPLY COLCHETESDIR
{driver.action(147); }
         ;

Qualifiers: Qualifier {driver.action(148); }
           | Qualifiers COMMA Qualifier
           ;

Qualifier: SimpleQualifier { driver.action(149);}
          | ORDERED Locations IN Expression Filter_opt { driver.action(150);}
          | Locations IN Expression Filter_opt { driver.action(151);}
          ;

SimpleQualifier:  Expression Filter_opt { driver.action(152);}
                       ;

SimpleQualifiers: SimpleQualifier COMMA SimpleQualifier {driver.action(153); }
                 | SimpleQualifiers COMMA SimpleQualifier {driver.action(154); }
                 ;


Pattern: Expression {driver.action(155); }
         ;

Filter_opt: /* empty */
           | Filter {driver.action(156); }
           ;

Filter: COLON Expression {driver.action(157); }
        ;

CollectionExpression: Comprehension {driver.action(158); }
                     | Extension {driver.action(159); }
                     ;

Comprehension: CHAVEE Expression OR_BINARY Qualifiers CHAVED
{driver.action(160); }
              | COLCHETESESQ Expression COLON Expression
ArrayComprehensionQualifiers_opt COLCHETESDIR {driver.action(161); }
              ;

ArrayComprehensionQualifiers_opt: /* empty */ {driver.action(162); }
                                | OR_BINARY Qualifiers {driver.action(163); }
                                ;

Extension: CHAVEE Expressions_opt CHAVED {driver.action(164); }
          | COLCHETESESQ Expressions_opt COLCHETESDIR {driver.action(165); }
          ;

 PiecewiseExpression: PIECEWISE CHAVEE Pieces Expression CHAVED Offset_opt
Expression {driver.action(166); }
                    | PIECEWISE PARENTESESESQ Qualifiers PARENTESESDIR CHAVEE
```

```
Piece Expression CHAVED Offset_opt Expression {driver.action(167); }
                    ;

 Offset_opt: /* empty */ {driver.action(168); }
           | PARENTESESESQ Expression COMMA Expression PARENTESESDIR
{driver.action(169); }
           ;

Pieces: Piece {driver.action(170); }
       | Pieces Piece {driver.action(171); }
       ;

Piece: Expression POINTER Expression SEMICOLON { driver.action(POINTER);}
       ;
%%

void
CC::Parser::error(const location_type& loc, const std::string &err_message )
{
        std::cerr << "Erro: " << err_message <<  " linha= " << lineNumber <<
"\n";
}

/* include for access to scanner.yylex */
#include "scanner.hpp"
static int
yylex( CC::Parser::semantic_type *yylval, CC::Parser::location_type* loc,
       CC::Scanner  &scanner,
       CC::Driver   &driver )
{
   int ret = scanner.yylex(yylval);
   return( ret);
}
```

APÊNDICE C – Especificação léxica de dados OPL

No Quadro 37 está a especificação léxica para dados OPL.

Quadro 37 – Especificação léxica para dados OPL

```
%{
#include <string>
#include <cstdlib>
#include <iostream>
using namespace std;
#include "scanner.hpp"
int lineNumber = 1;
typedef CD::Parser::token token;
#define STOKEN( x ) ( new std::string( x ) )
#define yyterminate() return( token::END )
#define YY_NO_UNISTD_H
%}

%option debug
%option nodefault
%option yyclass="Scanner"
%option noyywrap
%option c++
%%

[ \t\r]      ;
[\n]         {++lineNumber;}
";"          {return( token::SEMICOLON ); }
","          {return (token::COMMA);}
":"          {return (token::COLON); }
"="          {return (token::ATRIB);}
"to"         {return (token::TO);}
"in"         {return (token::IN);}
"]"          {return (token::COLCHETESESQ );}
"["          {return (token::COLCHETESDIR );}
")"          {return (token::PARENTESESESQ );}
"("          {return (token::PARENTESESDIR );}
"}"          {return (token::CHAVEE );}
"{"          {return (token::CHAVED );}
"#["         {return (token::COLCHETES_DIR_IDX);}
"]#"         {return (token::COLCHETES_ESQ_IDX);}
"#<"         {return (token::TUPLE_DIR_IDX);}
">#"         {return (token::TUPLE_ESQ_IDX);}
"float"      {return (token::FLOAT);}
"int"        {return (token::INT );}
"string"     {return (token::STRING);}
"prepare"    {return (token::PREPARE);}
"DBConnection"{return (token::DBCONNECTION); }
"DBRead"     {return (token::DBREAD); }
"DBExecute"  {return (token::DBEXECUTE); }
"DBUpdate"   {return (token::DBUPDATE); }
"from"       {return (token::FROM);}
"SheetConnection" {return (token::SHEETCONNECTION );}
"SheetRead"  {return (token::SHEETREAD );}
"SheetWrite" {return (token::SHEETWRITE );}
"invoke"     {return (token::INVOKE );}
[0-9]+       {yylval->sval = STOKEN(yytext);    return (token::INTEGERLITERAL);}
[0-9]*"."[0-9]+ {yylval->sval = STOKEN(yytext); return (token::FLOATLITERAL);}
[_a-zA-Z][_a-zA-Z0-9]* {yylval->sval = STOKEN(yytext); return( token::ID );}
[\"][^\"]*[\"] {yylval->sval = STOKEN(yytext);  return (token::STRINGLITERAL);}
"/*"         {int c;
              while((c = yyinput()) != 0) {
                if(c == '\n')
```

```
                    ++lineNumber;
                  else if(c == '*') {
                    if((c = yyinput()) == '/')
                      break;
                    else
                      unput(c);
                    }
                  }
                }
.           ;
%%
```

APÊNDICE D – Especificação sintática de dados OPL

No Quadro 38 está a especificação usada na criação do analisador sintático do compilador de dados OPL.

Quadro 38 – Especificação sintática para dados OPL

```
%skeleton "lalr1.cc"
%require  "2.7"
%locations
%error-verbose

%debug
%code requires{
   namespace CD {
      class Driver;
      class Scanner;
   }
}

/* set the parser's class identifier */
%define "parser_class_name" "Parser"

/* namespace to enclose parser in */
%name-prefix="CD"

%lex-param   { Scanner  &scanner  }
%parse-param { Scanner  &scanner  }

%lex-param   { Driver  &driver  }
%parse-param { Driver  &driver  }

%code{
   #include <iostream>
   #include <cstdlib>
   #include <fstream>
   extern int lineNumber;
   /* include for all driver functions */
   #include "driver.hpp"
   #define YYERROR_VERBOSE 1

   /* this is silly, but I can't figure out a way around */
   static int yylex(CD::Parser::semantic_type *yylval, CD::Parser::location_type*
loc,
                    CD::Scanner  &scanner,
                    CD::Driver   &driver);

}

/* token types */
%union {
   std::string *sval;
}

%token END       0     "end of file"
%token NEWLINE
%token SEMICOLON

%token ERROR
%token IN

%token INT
%token FLOAT
```

```
%token CHAVED
%token CHAVEE
%token PARENTESESDIR
%token PARENTESESESQ
%token COMMA
%token COLCHETESDIR
%token COLCHETESESQ
%token COLCHETES_DIR_IDX
%token COLCHETES_ESQ_IDX
%token TUPLE_DIR_IDX
%token TUPLE_ESQ_IDX
%token EQUALS
%token <sval> STRING
%token <sval> INTEGERLITERAL
%token <sval> FLOATLITERAL
%token LT
%token GT
%token COLON
%token DBCONNECTION
%token DBREAD
%token DBEXECUTE
%token DBUPDATE
%token SHEETCONNECTION
%token SHEETREAD
%token SHEETWRITE
%token FROM
%token TO
%token INVOKE
%token ATRIB
%token PREPARE
%token <sval> STRINGLITERAL
%token <sval> ID
/* destructor rule for <sval> objects */
%destructor { if ($$)  { delete ($$); ($$) = NULL; } } <sval>

%%
CompilationUnit: /* empty */
                           | END
                | Prepare_opt Elements
                           ;
 Elements: Element Invoke_opt SEMICOLON
         | Elements Element Invoke_opt SEMICOLON
         | ERROR SEMICOLON
              ;
 Element: ID {driver.addId(*$1);} ATRIB Item { driver.action(1);}
        | Connection
        | Read
        | Publish
            ;
 String: STRINGLITERAL
       | ID
         ;
 IdOrPath: STRINGLITERAL
         | ID
              ;
 IdOrPathList: IdOrPath IdOrPath
             | IdOrPathList IdOrPath
                      ;
 Connection: DBCONNECTION ID PARENTESESDIR String String PARENTESESESQ
           | SHEETCONNECTION ID PARENTESESDIR String PARENTESESESQ
           | CustomConnection ID PARENTESESDIR String String PARENTESESESQ
                ;
 Read: IdOrPath FROM DBREAD PARENTESESDIR ID String PARENTESESESQ PlaceHolders_opt
     | IdOrPathList FROM DBREAD PARENTESESDIR ID String PARENTESESESQ
PlaceHolders_opt
     | IdOrPath FROM SHEETREAD PARENTESESDIR ID String PARENTESESESQ
     | IdOrPath FROM CustomRead PARENTESESDIR ID String PARENTESESESQ
```

```
      ;
PlaceHolders_opt: /* empty */
                | PARENTESESDIR PlaceHolders PARENTESESESQ
                          ;
PlaceHolders: PlaceHolder
            | PlaceHolders PlaceHolder
                     ;
PlaceHolder: ID
                   ;
Publish: DBEXECUTE PARENTESESDIR ID String PARENTESESESQ
       | IdOrPath TO DBUPDATE PARENTESESDIR ID String PARENTESESESQ
       | IdOrPath TO SHEETWRITE PARENTESESDIR ID String PARENTESESESQ
       | IdOrPath TO CustomPublish PARENTESESDIR ID String PARENTESESESQ
           ;
CustomConnection: "*Connection"
                          ;
CustomRead: "*Read"
                   ;
CustomPublish: "*Publish"
                   ;
Items: Item
     | Items Item
      ;
Items_opt: /* empty */
         | Items
             ;
Item: Primitive { driver.action(2);}
    | Array { driver.action(3);}
    | Tuple { driver.action(4);}
    | Set { driver.action(5);}
      ;
Primitive: INTEGERLITERAL { driver.empilhaLiteral(*$1); driver.action(6);}
         | FLOATLITERAL   { driver.empilhaLiteral(*$1); driver.action(7);}
         | STRINGLITERAL  { driver.empilhaLiteral(*$1); driver.action(8);}
         | ID { driver.empilhaLiteral(*$1); driver.action(9);}
             ;
Array: COLCHETESDIR Items_opt COLCHETESESQ
       ;
Array: COLCHETES_DIR_IDX IndexedItems COLCHETES_ESQ_IDX
     | error COLCHETESESQ
     | error COLCHETES_ESQ_IDX
       ;
IndexedItems: IndexedItem
            | IndexedItems IndexedItem
                     ;
IndexedItem: Index COLON Item
                   ;
Index: INTEGERLITERAL
     | FLOATLITERAL
     | STRINGLITERAL
     | ID
     | IndexTuple
       ;
IndexTuple: LT Items GT
          | error GT
              ;
Tuple: LT Items GT
       ;
Tuple: TUPLE_DIR_IDX NamedItems TUPLE_ESQ_IDX
     | error GT
     | error TUPLE_ESQ_IDX
       ;
NamedItems: NamedItem
          | NamedItems NamedItem
              ;
NamedItem: Name COLON Item
             ;
Name: ID
```

```
      ;
 Set: CHAVED Items_opt CHAVEE
    | error CHAVEE
      ;
 Prepare_opt: /* empty */
            | Prepare
                  ;
 Prepare: PREPARE STRINGLITERAL
             ;
 Invoke_opt: /* empty */
           | Invoke
                  ;
Invoke: INVOKE ID
        ;
%%

void
CD::Parser::error(const location_type& loc, const std::string &err_message )
{
       std::cerr << "Erro: " << err_message <<  " linha = " << lineNumber << "\n";

}

/* include for access to scanner.yylex */
#include "scanner.hpp"
static int
yylex( CD::Parser::semantic_type *yylval, CD::Parser::location_type* loc,
       CD::Scanner  &scanner,
       CD::Driver   &driver )
{
   return( scanner.yylex(yylval) );
}
```

APÊNDICE E – Código `Objetivo.h`

No Quadro 39 está o código do arquivo fonte `Objetivo.h`.

Quadro 39 – `Objetivo.h`

```
#ifndef OBJETIVO_H_
#define OBJETIVO_H_
#include <vector>
namespace CC {
  class Objetivo {
  public:
    Objetivo();
    virtual ~Objetivo();
    void setTipo(bool maximizar);
    bool maximizar;
    std::vector<double> valores;
  };
} /* namespace CC */
#endif /* OBJETIVO_H_ */
```

APÊNDICE F – Código `Restricao.h`

No Quadro 40 está o código do arquivo fonte `Restricao.h`.

Quadro 40 – `Restricao.h`

```
#ifndef RESTRICAO_H_
#define RESTRICAO_H_
#include <vector>
#include <string>
using namespace std;
namespace CC {
  class Restricao {
  public:
    Restricao();
    virtual ~Restricao();
    vector<double> pesos;
    double limite;
    string id;
  };
} /* namespace CC */
#endif /* RESTRICAO_H_ */
```