

Departamento de Engenharia de Computação e Sistemas Digitais

**PCS 2056 – Linguagens e Compiladores**

**Construção do compilador de Kipple para CIL**

10/12/2011

Bruno Pezzolo dos Santos 5948816

Carla Guillen Gomes 5691366

**Sumário**

[1. Introdução 2](#_Toc311229505)

[2. Especificação da linguagem fonte: Kipple 3](#_Toc311229506)

[2.1. Kipple em descrição Wirth 3](#_Toc311229507)

[2.2. Descrição Reduzida 4](#_Toc311229508)

[3. Análise Léxica 5](#_Toc311229509)

[3.1. Autômatos 6](#_Toc311229510)

[3.2. Implementação 8](#_Toc311229511)

[4. Análise Sintática 11](#_Toc311229512)

[4.1. Submáquinas do Autômato de Pilha Estruturado (APE) 11](#_Toc311229513)

[4.2. Implementação 13](#_Toc311229514)

[5. Tradução de Comandos 15](#_Toc311229515)

[5.1. Ambiente de Execução 16](#_Toc311229516)

[6. Análise Semântica 19](#_Toc311229517)

[6.1. Ações semânticas 19](#_Toc311229518)

[7. Testes 22](#_Toc311229519)

[8. Conclusão 24](#_Toc311229520)

# Introdução

Este documento descreve o compilador desenvolvido pelos alunos Bruno Pezzolo dos Santos e Carla Guillen Gomes na segunda avaliação da disciplina PCS2056 – Linguagens e Compiladores. O compilador foi desenvolvido com fins didáticos, sendo desconsideradas características como otimização de código-fonte e desempenho.

O compilador foi escrito na linguagem C, tem como código fonte a linguagem Kipple e gera código objeto em assembly Common Intermediate Language (CIL), uma linguagem orientada a objetos e baseada em pilha. Seu bytecode pode ser traduzido diretamente para código de máquina ou executado por uma máquina virtual.

A arquitetura do compilador é orientada à sintaxe, ou seja, o módulo de análise sintática gerencia as atividades através de requisições aos outros módulos, conforme representação a seguir:

Código fonte

Tokens

Código objeto

Analisador Sintático

Analisador Léxico

Tabela de símbolos

Solicita token

Analisador Semântico

O desenvolvimento deste compilador foi inteiramente baseado no código do compilador desenvolvido previamente na mesma disciplina que gera código objeto em MVN.

# Especificação da linguagem fonte: Kipple

A linguagem de alto nível a ser compilada é a Kipple, uma linguagem minimalista e baseada em pilha, cuja especificação pode ser vista em <http://www.esolangs.org/files/kipple/doc/kipple03.html>

Um exemplo de programa está representado a seguir:

|  |
| --- |
| 1 > a # push 1 na pilha a  2 > a # push 2 na pilha a  b < a # pop em a e push resultado em b  a? # limpa pilha a caso o elemento no topo seja 0 |

## Kipple em descrição Wirth

|  |
| --- |
| program = { operation | loop } .  number = digit {digit} .  digit = “0” | “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” | “9” .  id = “a” | “b” | “c” | “d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "@” .  operation = id (operation\_right | operation\_left | operation\_unary) | number operation\_right .  operation\_right = ">" ( id [operation\_right | operation\_left | operation\_unary] ) .  operation\_left= ( "<" | "+" | "-" ) (id [operation\_right | operation\_left | operation\_unary] | number [operation\_right] ) .  operation\_unary = “?” .  loop = “(“ (id | operation) {operation | loop} “)”. |

## Descrição Reduzida

Para que a análise sintática fosse implementada usando um autômato de pilha estruturado, a descrição em notação de Wirth foi reduzida através do agrupamento de alguns não-terminais. O resultado da redução é mostrado a seguir.

|  |
| --- |
| program = { id (operation\_right | operation\_left | "?") | number operation\_right | loop } .  operation\_right = ">" id [operation\_right | operation\_left | "?"] .  operation\_left= ( "<" | "+" | "-" ) ( id [operation\_right | operation\_left | "?"] | number [operation\_right] ).  loop = "(" (id [operation\_right | operation\_left | "?"] | number operation\_right) {id (operation\_right | operation\_left | "?") | number operation\_right | loop} ")".  number = digit {digit} .  digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .  id = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z" | "@" . |

# Análise Léxica

O módulo de análise léxica é responsável por converter o texto fonte em componentes chamados *tokens* ou átomos. Fornecendo, então, uma representação mais adequada para realização da análise e tradução pelos outros módulos do compilador.

Dessa maneira, o analisador léxico efetua a leitura do código fonte, e com o conhecimento das especificações da linguagem agrega os caracteres lidos do código em unidades que contenham um sentido completo para a linguagem, os *tokens*, descartando os caracteres que não possuem significado para a linguagem (por exemplo, caracteres de tabulação, na linguagem C).

Porém, mais informações são necessárias para as análises subsequentes, como o tipo de informação que está sendo representada pelo *token* (como, por exemplo, a classificação do átomo como um "identificador", "palavra reservada" ou "número"), além disso, um valor para o *token* também é necessário em outras etapas da compilação, esse valor pode ser um valor númerico do *token*, um índice em uma tabela de símbolos ou o próprio texto. Sendo também papel do analisador léxico realizar essas tarefas.

A tabela abaixo mostra os tipos de átomos reconhecidos pelo analisador léxico, definidos através de expressões regulares.

|  |  |
| --- | --- |
| **Classe do token** | **Expressão Regular** |
| Identificador de pilha | "A" | … |  "Z" | "a" | … |  "z" | "@" |
| Número | [dígito][dígito]\* |
| Operador | "+" | "-" | ">" | "<" | "?" |
| Caractere Especial | "(" | ")" |
| Espaçador | space | \n | \t |
| Comentário | [#][qualquer\_caractere\_ascii]\*[\n] |

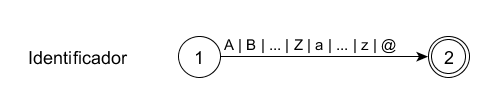
O analisador léxico implementado realiza uma simples detecção de erros, retornando *tokens* inválidos quando o conjunto de caracteres lidos não corresponde a nenhuma das expressões representadas acima.

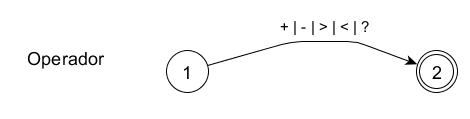
A linguagem Kipple define que os identificadores de pilha são *case insensitive*. Portanto, o compilador deve tratar identificadores escritos em letra maiúscula e minúscula igualmente. Durante a análise léxica, os tokens do tipo identificador de pilha são normalizados para que todos sejam representados com letra minúscula, facilitando assim o trabalho do analisador semântico.

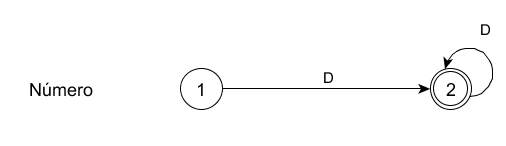
Uma observação é que o transdutor implementado ignora os tokens do tipo espaçador e comentário. Portanto, tokens destes tipos não são repassados ao analisador sintático.

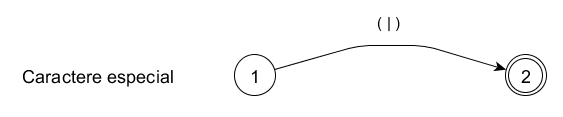
## Autômatos

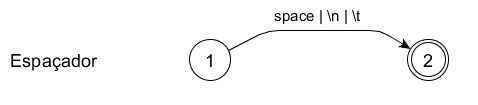
Para facilitar a representação do reconhecimento dos *tokens* pelo analisador léxico, as expressões regulares foram convertidas em autômatos finitos equivalentes, representados a seguir, onde:

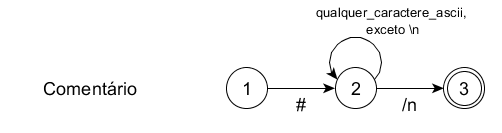




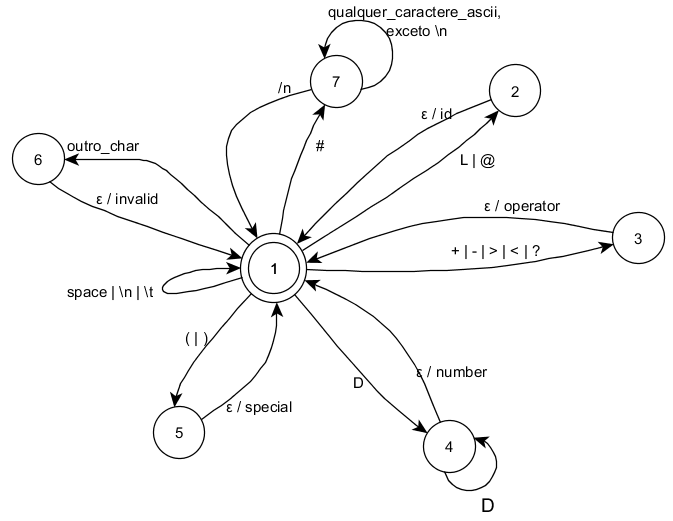








Estes autômatos reconhecem as correspondentes linguagens por eles definidas. Foi feito um agrupamento de todos os autômatos e, por fim, o autômato foi convertido em transdutor, que emite como saída o átomo encontrado ao abandonar cada um dos estados finais para iniciar o reconhecimento de mais um átomo do texto.

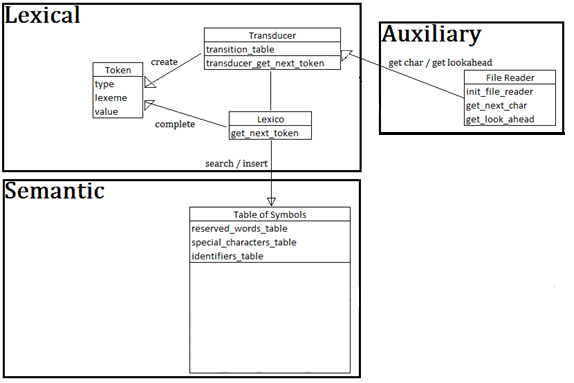


onde:  
L = A | B | … | Z | a | … | z  
D = 0 | 1 | .. | 9

A partir deste transdutor é possível observar que não há tokens para os tipos espaçadores e comentários. O analisador léxico ignora estes tipos conforme explicado anteriormente.

## Implementação

O analisador léxico foi implementado como uma sub-rotina que é chamada pelo analisador sintático sempre que um novo *token* for necessário. A arquitetura básica da implementação do analisador léxico é a que segue:



reserved\_words\_table

special\_characters\_table

id\_table

search\_id\_table

search\_special\_table

search\_operator\_table

display\_id\_table

display\_special\_table

display\_operator\_table

Os arquivos token.h e token.c representam um *token* na linguagem, contendo seu tipo, lexema (ou seja, o texto que representa o *token*) e seu valor. Neles são definidos todos os tipos possíveis de tokens existentes na linguagem. Vale observar que para alguns tipos () o atributo *lexeme* é utilizado no lugar do *value*.

Os arquivos lexico.h e lexico.c representam o analisador léxico. Na implementação realizada, o analisador léxico faz requisições de tokens para o transdutor, que retorna um token com um tipo inicial. É então, papel do analisador léxico completar o token retornado com o restante dos dados necessários, ou seja, calcular o valor dos tokens, bem como fazer o tratamento necessário para a característica *case insensitive* dos identificadores da linguagem.

Os arquivos table\_of\_symbols.h e table\_of\_symbols.c representam as tabelas de símbolos existentes no compilador. Três tabelas de símbolos são utilizadas, a de operadores, a de caracteres especiais e a de identificadores. Por terem tamanho fixo, as tabelas foram implementadas através de matrizes, inicializadas com todos os lexemas possíveis. Inclusive a tabela de identificadores, pois na linguagem Kipple só há possibilidade de 27 variáveis (a-z e @).

Os arquivos file\_reader.c e file\_reader.h implementam rotinas auxiliares de leitura. Dado um arquivo de entrada há funções para acesso ao caractere sendo lido e ao próximo caractere a ser lido, essas funções são get\_next\_char() e get\_look\_ahead(). O arquivo de entrada é determinado ao inicializar o file\_reader através da função void init\_file\_reader(char \*path). Esta inicialização é feita na função *main*.

Os arquivos transducer.c e transducer.h implementam o transdutor mostrado no Capítulo 3.1. Para isto, existe uma matriz de transição (denominada transition\_table) onde as linhas representam os estados atuais e, dado um caractere de entrada (representado pelas colunas), o valor da célula correspondente indica o próximo estado. Os estados são representados por números inteiros e correspondem aos estados da figura do transdutor. Há uma rotina para inicialização desta tabela, um trecho desta rotina é mostrado abaixo.

|  |
| --- |
| void init\_transition\_table() {  int i, j;  for (i = 1; i < NUM\_STATES + 1; i++)  for (j = 0; j < NUM\_TYPE\_CHARS; j++)  transition\_table[i][j] = 0;  /\* TTYPE\_ID \*/  transition\_table[1][*letter*] = 2;  transition\_table[1][*at*] = 2;  /\* TTYPE\_OPERATOR \*/  transition\_table[1][*operator*] = 3;  /\* TTYPE\_NUM \*/  transition\_table[1][*digit*] = 4;  transition\_table[4][*digit*] = 4;  /\* TTYPE\_SPCEIAL \*/  transition\_table[1][*special*] = 5;  /\* space, \t and \n \*/  transition\_table[1][*space*] = 1;  transition\_table[1][*linebreak*] = 1; |

A função transducer\_get\_next\_token() percorre o arquivo de entrada (através de funções auxiliares) e, a cada caractere lido, atualiza seu estado. Quando o transdutor volta para o estado 1 (inicial), ele atualiza uma variável global token preenchendo seu *type* e *lexeme* (o preenchimento do campo *value* e eventual tratamento do lexema é feito posteriormente por rotinas do léxico). Para possibilitar as transições vazias do transdutor é usado o caractere look\_ahead. Quando, usando o lookahead, é observado que o token incompleto se tornaria inválido, o token é atualizado e o transdutor volta ao estado inicial.

Um teste foi realizado comprovando o correto funcionamento do analisador léxico. A entrada e saída correspondente são mostradas a seguir.

|  |  |
| --- | --- |
| **Entrada\_teste.k** | |
| 1 > a # push 1 na pilha a  2 > a # push 2 na pilha a  b < A # pop em a e push resultado em b  a? # limpa pilha a caso o elemento no topo seja 0  (a>b) | |
| **Saída** | **Tabelas de símbolos** |
| tokens reconhecidos no formato: lexeme  TYPE (value) |  |

# Análise Sintática

A análise sintática é responsável por verificar se o código a ser analisado corresponde à gramática da linguagem-fonte.

Em compiladores orientados por sintaxe, como é o caso do compilador desenvolvido, o analisador sintático é responsável por controlar as atividades do compilador, sendo responsável por efetuar chamadas de funções léxicas para o recebimento de *tokens* conforme os mesmos são consumidos.

A partir dos *tokens* gerados pelo analisador léxico, o analisador sintático analisa a sequência de recebimentos dos *tokens* e monta a árvore sintática.

## Submáquinas do Autômato de Pilha Estruturado (APE)

A implementação do analisador sintático do compilador desenvolvido foi baseada no autômato de pilha estruturado (APE) tornando-se necessária a tradução dos quatro terminais (PROGRAM, OPERATION\_RIGHT, OPERATION\_LEFT e LOOP) em autômatos.

Para esta tradução, foi utilizado o gerador de autômatos disponível em: <http://radiant-fire-72.heroku.com/>. Ao inserir descrição reduzida da linguagem na notação Wirth, mostrada no Capítulo 2.3, obteve-se as seguintes listas de transições, com seus respectivos autômatos:

|  |  |
| --- | --- |
| **PROGRAM** | |
| initial: 0  final: 3  (0, id) -> 1  (0, number) -> 2  (0, loop) -> 3  (1, operation\_right) -> 3  (1, operation\_left) -> 3  (1, "?") -> 3  (2, operation\_right) -> 3  (3, id) -> 1  (3, number) -> 2  (3, loop) -> 3 | https://lh3.googleusercontent.com/FOLNIOnJEhmSYgJO_qcsft2CUDviBE6ronLJkQKNN1AUzWQwFTuJW5j8vJyyeMHNjyo19Dl7TyOLO0y98O61GTP2xOKHj53IpfMQDOsvG4DhLV03DSM |

|  |  |
| --- | --- |
| **OPERATION\_RIGHT** | |
| initial: 0  final: 2, 3  (0, ">") -> 1  (1, id) -> 2  (2, operation\_right) -> 3  (2, operation\_left) -> 3  (2, "?") -> 3 | https://lh3.googleusercontent.com/lcDcJ0EIBTZsq_lXSXOvO69BS7F5YS7bleZuVfZW0va6g14ecryiAfUTYLzI-OKgAd-eghkrywUv51pDDOC30brYoFsr9-n6tmf1MjTB3lpeW0HRmlE |
| **OPERATION\_LEFT** | |
| initial: 0  final: 2, 3, 4  (0, "<") -> 1  (0, "+") -> 1  (0, "-") -> 1  (1, id) -> 2  (1, number) -> 3  (2, operation\_right) -> 4  (2, operation\_left) -> 4  (2, "?") -> 4  (3, operation\_right) -> 4 | https://lh5.googleusercontent.com/X01_ZG0XCsQk9GZ2ngktKR01XrZTSYyujXu0kCalPGAMDT8t0C7CAu08IaF6ZEPNSCh6HuALezpGKnkxGPW1B4gqLYzE_aUyMi6mV1ci_NP2wc_q6JU |
| **LOOP** | |
| initial: 0  final: 6  (0, "(") -> 1  (1, id) -> 2  (1, number) -> 3  (2, id) -> 4  (2, operation\_right) -> 5  (2, operation\_left) -> 5  (2, "?") -> 5  (2, number) -> 3  (2, loop) -> 5  (2, ")") -> 6  (3, operation\_right) -> 5  (4, operation\_right) -> 5  (4, operation\_left) -> 5  (4, "?") -> 5  (5, id) -> 4  (5, number) -> 3  (5, loop) -> 5  (5, ")") -> 6 | https://lh3.googleusercontent.com/VFqWvZfZm9_TiHuZsPvzeikbD0rxuCW9Y8BsIIpsto29ZKWl-vonQ2j5rxV3zZlPS1UTjps2-8VhJaxKlplJQjcSFRquaR4rbZStfIhfbjpdPsJjkaA |

## Implementação

O analisador sintático foi implementado como uma rotina do programa principal. A princípio, são inicializados todos os recursos utilizados pela análise sintática (o analisador léxico, tabela de transição e o autômato de pilha estruturado).

Então, a rotina pede o próximo *token* ao analisador léxico e, enquanto não terminar de ler o arquivo ou encontrar um erro na análise sintática, o analisador executa um passo do seu autômato de pilha estruturado. O passo do APE é sua principal função, ilustrada abaixo:

|  |
| --- |
| int spa\_step() {  int machine\_token\_type = spa\_convert\_token\_to\_machine\_type();  if (machine\_token\_type == MTTYPE\_INVALID) return 0;  return transition\_current\_machine\_with\_token(machine\_token\_type);  } |

O autômato de pilha estruturado (APE) utiliza autômatos finitos que representam os quatro não terminais da linguagem. Primeiramente ele converte o *token* retornado pelo analisador sintático, para que ele seja reconhecido nas transições dos autômatos finitos da linguagem.

Então, o passo do APE realiza uma transição da máquina atual com o *token* convertido. Essa transição possui três possibilidades. O primeiro caso consiste na existência de uma transição na máquina corrente do estado atual com o *token* recebido. Esse caso é tratado na função “transition\_to\_next\_state(int next\_state)”:

|  |
| --- |
| void transition\_to\_next\_state (int next\_state) {  actions\_on\_state\_transition[current\_machine.machine\_id]   [current\_machine.current\_state]  [spa\_convert\_token\_to\_machine\_type()](token);  current\_machine.current\_state = next\_state;} |

Nesse caso, o mais simples deles, apenas ocorre uma mudança de estado da máquina atual. Além disso, a transição também desencadeia uma ação semântica, com uma de chamada de função guardada na matriz actions\_on\_state\_transition. As ações semânticas serão descritas com maiores detalhes no capítulo referente à análise semântica.

O segundo caso, é quando a transição de estado é uma chamada de submáquina. Esse caso é tratado na seguinte função:

|  |
| --- |
| void call\_machine(int machine\_type) {  actions\_on\_machine\_transition[current\_machine.machine\_id]  [current\_machine.current\_state]  [machine\_type](token);  current\_machine.current\_state = current\_machine.machine\_calls  [current\_machine.current\_state]  [machine\_type];  spa\_stack\_push(current\_machine, spa\_stack);  current\_machine = machines\_array[machine\_type];  current\_machine.current\_state = 0;  } |

Para que seja possível o correto funcionamento do APE, é necessário que se guarde a máquina atual e o seu estado corrente em uma pilha. Para isso, foi implementada uma pilha em “*spa\_stack.h*” e “*spa\_stack.c*”, utilizada pelo APE para empilhar a máquina corrente antes de transicionar para a próxima. Além disso, a transição também desencadeia uma ação semântica.

O último caso existente é aquele em que a o autômato chegou em seu estado final, mas ainda existem máquinas empilhadas. Nesse caso, o que ocorre é um retorno à máquina anterior, pela função:

|  |
| --- |
| void return\_machine() {  actions\_on\_machine\_return[current\_machine.machine\_id]  [current\_machine.current\_state](token);  current\_machine = spa\_stack\_pop(spa\_stack); } |

Nesse caso, é desencadeada uma ação semântica, e, além disso, retorna-se a máquina que se encontra no topo da pilha, para continuar a execução do APE de maneira adequada.

Todas as informações das máquinas utilizadas pelo APE se encontram nos arquivos “*machines.h*” e “*machines.c*”. Sendo estes modelos das máquinas, com suas tabelas de transição de estados, estados finais e transições que representam chamadas de máquinas.

# Tradução de Comandos

Antes da implementação do analisador semântico, traduções de algumas estruturas e comandos foram feitas de forma a definir um padrão e auxiliar o desenvolvimento das ações semânticas. Tais traduções serão expostas neste capítulo.

|  |  |
| --- | --- |
| Linguagem de entrada | CIL |
| 12>a  OU  a>12 | // load variavel a  ldloc a  // load constante 12  ldc.i4 12  box [mscorlib]System.Int32  // push (instance = a e object = 12)  callvirt instance void [mscorlib]System.Collections.Stack::Push(object) |
| b>a  OU  a<b | ldloc a // load variavel a  ldloc b // load variavel b  // pop em a (e joga objeto retornado na stack)  callvirt instance object [mscorlib]System.Collections.Stack::Pop()  // push (instance = a e object = b.pop())  callvirt instance void [mscorlib]System.Collections.Stack::Push(object) |
| a+2 | ldloc a //preload  ldloc a  //peek em a (e joga objeto retornado na stack)  callvirt instance object [mscorlib]System.Collections.Stack::Peek()  tunbox.any [mscorlib]System.Int32 //desconverte  ldc.i4 2  add  box [mscorlib]System.Int32 //converte para object  //push (instance = a(preloaded) e object = resultado do add)  callvirt instance void [mscorlib]System.Collections.Stack::Push(object) |
| a? | ldloc a  callvirt instance object [mscorlib]System.Collections.Stack::Peek()  ldc.i4 0  box [mscorlib]System.Int32  bne.un \_ENDCLEAR  ldloc a  callvirt instance void [mscorlib]System.Collections.Stack::Clear()  \_ENDCLEAR: |
| (a a>b) | \_LOOP\_0:  ldloc a  callvirt instance object [mscorlib]System.Collections.Stack::Peek()  unbox.any [mscorlib]System.Int32  ldc.i4 0  beq \_LOOP\_0\_F  ldloc b  ldloc a  callvirt instance object [mscorlib]System.Collections.Stack::Pop()  callvirt instance void [mscorlib]System.Collections.Stack::Push(object)  br \_LOOP\_0  LOOP\_0\_F: |

## Ambiente de Execução

Como é possível observar através dos exemplos acima, é possível utilizar classes da linguagem C# no desenvolvimento de programas utilizando CIL. Como a linguagem Kipple é baseada em pilhas, foi utilizada a classe System.Collections.Stack da linguagem C#.

Dada a característica da linguagem Kipple em que, quando realiza-se um POP em uma pilha vazia, ela deve retornar o valor 0, para replicar esta funcionalidade, uma nova classe foi implementada. A nova classe, herdada da classe Stack, recebeu o nome de SpecialStack. Ela sobrescreve os métodos Peek() e Pop() para que não lance uma exceção quando a pilha for vazia, mas sim, retorne o valor 0.

|  |
| --- |
| namespace Auxiliar  {  public class SpecialStack<T> : System.Collections.Stack  {  public override Object Pop() {  try  {  return base.Pop();  }  catch (InvalidOperationException e)  {  return 0;  }  }  public override Object Peek()  {  try  {  return base.Peek();  }  catch (InvalidOperationException e)  {  return 0;  }  }  }  } |

A implementação foi feita em alto nível (ver quadro acima) e o código CIL gerado foi adicionado ao ambiente de execução.

O ambiente de execução é fornecido pelo ambiente C# e a principal biblioteca utilizada é a mscorlib.

No início da compilação, o código presente no arquivo *ambienteexecucao.il* é copiado para o arquivo de saída. Este código contém a classe auxiliar SpecialStack e a inicialização do programa compilado, contendo a declaração das 27 variáveis possíveis e suas inicializações. Um trecho deste código é mostrado a seguir.

|  |
| --- |
| .assembly extern mscorlib  {}  .assembly App  {}  // == AUXILIAR CLASS ===  // special stack returns 0 when Pop() or Peek() is called for an empty stack  .class public auto ansi beforefieldinit Auxiliar.SpecialStack<T>  extends [mscorlib]System.Collections.Stack  {  .method public hidebysig virtual instance object  Pop() cil managed  {  …  } // end of method SpecialStack::Pop  .method public hidebysig virtual instance object  Peek() cil managed  {  …  } // end of method SpecialStack::Peek  …  } // end of class Auxiliar.SpecialStack  // =============== CLASS MEMBERS DECLARATION ===================  .class private auto ansi beforefieldinit ConsoleApplication2.Program  extends [mscorlib]System.Object  {  .method private hidebysig static void Main(string[] args)  {  .entrypoint  .maxstack 10  .locals init ([0] class Auxiliar.SpecialStack<int32> a,  [1] class Auxiliar.SpecialStack<int32> b,  [2] class Auxiliar.SpecialStack<int32> c,  …  [26] class Auxiliar.SpecialStack<int32> @  )    // cria as pilhas  newobj instance void class Auxiliar.SpecialStack<int32>::.ctor()  stloc a  newobj instance void class Auxiliar.SpecialStack<int32>::.ctor()  stloc b  …  newobj instance void class Auxiliar.SpecialStack<int32>::.ctor()  stloc z  newobj instance void class Auxiliar.SpecialStack<int32>::.ctor()  stloc @  // -- aqui começa a área de código |

Com a cópia do conteúdo do arquivo *ambienteexecucao.il*  para o arquivo de saída, a classe SpecialStack torna-se disponível para uso, assim como as funcionalidades da biblioteca mscorlib.

# Análise Semântica

A análise semântica do compilador desenvolvido consiste nas ações semânticas desencadeadas pelas transições do APE. As ações semânticas consistem em imprimir no arquivo de saída o código objeto gerado, ou realizar atualizações nos estados internos do compilador, conforme os *tokens* são lidos e o APE é transicionado.

## Ações semânticas

Nesta seção serão descritas as ações semânticas desenvolvidas. As mesmas se encontram no arquivo semantic\_actions.c e são chamadas através de consultas às matrizes actions\_on\_state\_transition, actions\_on\_machine\_return e actions\_on\_machine\_return. Estas matrizes mapeiam as transições do APE com as ações semânticas correspondentes. As matrizes são preenchidas no início da compilação através da função init\_semantic\_actions.

**dummy\_semantic\_action**

Esta ação semântica não gera código nem modifica variáveis. É uma ação executada em todas as transições do APE que não possuem ações definidas.

**get\_loop\_label**

Esta ação semântica retorna um rótulo utilizado nos loops. O rótulo possui formato \_LOOP\_<loop\_counter> e o loop\_counter é incrementado sempre que esta ação semântica é executada.

**end\_program**

Esta ação semântica gera código para encerramento do código gerado, imprimindo o comando ret e caracteres } que fecham escopos abertos anteriormente.

**start\_loop**

Esta ação semântica é chamada na transição do estado 1 ao receber um id na submáquina LOOP. Para permitir loops aninhados, é disponibilizada uma pilha command\_operand\_stack onde os rótulos dos loops são empilhados nesta ação semântica. Esta ação também imprime no arquivo de saída código referente ao início de um loop.

**end\_loop**

Esta ação semântica é chamada ao final de um LOOP e imprime no arquivo de saída código referente ao fim de um loop, utilizando para isso o rótulo extraído da pilha command\_operand\_stack.

**save\_operand**

Salva o operando lido na variável operand1 ou operand2 (caso a operand1 já esteja "ocupada"). Esta ação é chamada sempre que uma constante ou identificador de pilha é lido.

**save\_operator**

Salva o operador lido na constante operator1. É chamada sempre que é lido um dos caracteres: >, < , + ou -

**resolve\_greater\_than**

Esta ação imprime no código de saída uma operação de push no operando2 e pop no operando1 (ou carrega a constante, caso o operando1 não seja um identificador). Seu código pode ser visto a seguir:

|  |
| --- |
| **void** **resolve\_greater\_than**() {  **sprintf**(buffer, "\t\tldloc %s\n", operand2);  write\_to\_code(buffer);  **if**(is\_identifier(operand1)) {  **sprintf**(buffer, "\t\tldloc %s\n", operand1);  write\_to\_code(buffer);  **sprintf**(buffer,"\t\tcallvirt instance object [mscorlib]System.Collections.Stack::Pop()\n");  write\_to\_code(buffer);  }  **else** {  **sprintf**(buffer, "\t\tldc.i4 %s\n\t\tbox [mscorlib]System.Int32 \n", operand1);  write\_to\_code(buffer);  }  **sprintf**(buffer,"\t\tcallvirt instance void [mscorlib]System.Collections.Stack::Push(object)\n\n");  write\_to\_code(buffer);  } |

**resolve\_less\_than**

Similar à ação resolve\_greater\_than, com a diferença na ordem em que os operandos são carregados.

**resolve\_plus**

Ação responsável por escrever no código de saída as operações referentes à adição. Os operandos foram carregados previamente e estão armazenados nas variáveis locais operando1 e operando2.

**resolve\_minus**

Similar à ação resolve\_greater\_than, onde o comando add é substituído pelo comando sub.

**resolve\_question\_mark**

Ação responsável por escrever no código de saída as operações referentes ao comando Kipple clear. O operando foi carregado previamente através da ação save\_operand.

**resolve\_operation\_and\_continue**

Ação responsável por chamar outra ação semântica, que resolverá a operação com os operandos armazenados nas variáveis operand1 e operand2. Depois, copia o conteúdo do operand2 na variável operand1. Através desta ação é possível encadeamentos do tipo a>b<c.

**resolve\_operation**

Similar à resolve\_operation\_and\_continue, mas ao final, faz com que operand1 e operand2 sejam nulos.

# Testes

O seguinte teste mostra o correto funcionamento das diversas funcionalidades do compilador desenvolvido.

|  |
| --- |
| **Código de entrada** |
| a<1  a<2  3>a  (a a>b)  b+3  a-b  B? |
| **Código compilado** |
| <código do ambiente de execução omitido>  ldloc a  ldc.i4 1  box [mscorlib]System.Int32  callvirt instance void [mscorlib]System.Collections.Stack::Push(object)  ldloc a  ldc.i4 2  box [mscorlib]System.Int32  callvirt instance void [mscorlib]System.Collections.Stack::Push(object)  ldloc a  ldc.i4 3  box [mscorlib]System.Int32  callvirt instance void [mscorlib]System.Collections.Stack::Push(object)  \_LOOP\_0: ldloc a  callvirt instance object [mscorlib]System.Collections.Stack::Peek()  unbox.any [mscorlib]System.Int32  ldc.i4 0  beq \_LOOP\_0\_F  ldloc b  ldloc a  callvirt instance object [mscorlib]System.Collections.Stack::Pop()  callvirt instance void [mscorlib]System.Collections.Stack::Push(object)  br \_LOOP\_0  \_LOOP\_0\_F:  ldloc b  ldloc b  callvirt instance object [mscorlib]System.Collections.Stack::Peek()  unbox.any [mscorlib]System.Int32  ldc.i4 3  add  box [mscorlib]System.Int32  callvirt instance void [mscorlib]System.Collections.Stack::Push(object)  ldloc a  ldloc a  callvirt instance object [mscorlib]System.Collections.Stack::Peek()  unbox.any [mscorlib]System.Int32  ldloc b  callvirt instance object [mscorlib]System.Collections.Stack::Pop()  unbox.any [mscorlib]System.Int32  sub  box [mscorlib]System.Int32  callvirt instance void [mscorlib]System.Collections.Stack::Push(object)  ldloc b  callvirt instance object [mscorlib]System.Collections.Stack::Peek()  ldc.i4 0  box [mscorlib]System.Int32  bne.un \_ENDCLEAR  ldloc b  callvirt instance void [mscorlib]System.Collections.Stack::Clear()  \_ENDCLEAR:  ret |

# Conclusão

Os objetivos foram alcançados com sucesso, conforme demonstra o teste do capítulo anterior. O compilador cumpre todos os requisitos solicitados:

* Permite operações do tipo:
  + >
  + <
  + +
  + -
* Permite loops (inclusive aninhados)

A única funcionalidade da linguagem Kipple que não foi implementada foi a produção de E/S, pois a mesma não era obrigatória. Porém, com o conhecimento adquirido ao longo do desenvolvimento percebe-se que a implementação é trivial, através de comandos do tipo:

|  |
| --- |
| call string[mscorlib]System.Console::ReadLine()  call int32[mscorlib]System.Int32::Parse(string)  call void [mscorlib]System.Console::WriteLine(int32) |