Projeto do Compilador

Curso Cooperativo de Engenharia da Computação 2007 – Escola Politécnica da USP

Fabio Correia Kung

Nataniel Simon

1. O Projeto

Já tem algum tempo que existe o interesse da equipe em estudar a plataforma Java um pouco mais a fundo. Especificamente, seria interessante estudar com profundidade como funciona a máquina virtual Java.

Um dos passos primordiais para tal, é entender o formato dos bytecodes que a máquina virtual Java interpreta (e compila através do JIT). São chamados de bytecodes, pois cada opcode ocupa apenas um byte (sem contar os operandos).

Aproveitando a oportunidade na construção deste compilador, a equipe decidiu por fazer com que a saída do compilador fossem bytecodes Java válidos. Em outras palavras, a saída do compilador é uma classe Java executável, que se comporta como descrito pelo programa na linguagem especificada para a entrada do compilador.

Recentemente, um de nós também tem tido um interesse muito grande pela linguagem de programação Ruby, e esta foi uma grande oportunidade de aprofundar o conhecimento na linguagem. Desta forma, o compilador foi implementado em Ruby, uma linguagem orientada a objetos, sem tipagem explícita, bastante dinâmica (classes abertas que permitem a inclusão de métodos novos em tempo de execução, como SmallTalk) e diversas características funcionais, inspiradas pelo LISP.

Cada uma das seções a seguir, detalha o processo de desenvolvimento do compilador e as decisões de projeto associadas a cada tarefa.

1.1. Linguagem de entrada adotada

A construção do compilador tem fins didáticos, portanto foi escolhida uma linguagem simples (e também didática) para ser compilada. A linguagem escolhida foi a criada por Niklaus Wirth: PL/0, apresentada em aula, com leves modificações.

A linguagem é bastante simples, só trabalha com inteiros, suporta declaração de procedimentos, tem controle de escopo básico, já que as variáveis são visíveis por todos os escopos mais internos e não conta com operações de entrada e saída.

Dado o tempo reduzido para dedicação ao projeto, durante o desenvolvimento nos vimos diversas vezes no dilema: “vale a pena introduzir mais funcionalidades na linguagem?”. A conclusão foi que maioria das funcionalidades interessantes para serem adicionadas não agregariam muito valor ao aprendizado do processo de compilação, a linguagem da forma que estava já servia bem para o propósito de construção de um compilador que contemplasse os aspectos vistos durante a disciplina.

Portanto, sempre que nos víamos neste dilema a decisão levou em conta o foco e a vontade de aprender os bytecodes da plataforma Java, ou seja, ao invés de complicar a linguagem de entrada, decidimos por gerar bytecodes da forma aceita pela especificação da máquina virtual Java.

A linguagem PL/0 foi levemente modificada, para integrar um aspecto crítico: entrada e saída. Segue a gramática (na notação de Wirth) adotada para o projeto:

```
program = block "." .

block = [ "CONST" ident "=" number {"," ident "=" number} ";"]
```

```
        [ "VAR" ident {"," ident} ";"]
        { "PROCEDURE" ident ";" block ";" } statement .

statement = [ ident ":=" expression |
            "CALL" ident |
            "OUT" expression |
            "IN" ident |
            "BEGIN" statement {";" statement } "END" |
            "IF" condition "THEN" statement |
            "WHILE" condition "DO" statement ] .

condition = expression ("="|"#"|"<"|"<="|">"|">=") expression .

expression = [ "+"|"-"] term { ("+"|"-") term} .

term = factor {("*"|"/") factor} .

factor = ident | number | "(" expression ")" .
```

A principal modificação foi na regra de produção para os “statement”s, onde foram inclusas duas novas instruções: `IN` e `OUT`.

1.2. Analisando a linguagem

Da linguagem adotada para o projeto, puderam ser identificados os possíveis tokens para a construção do Analisador Léxico. O conjunto de tokens está representado a seguir:

```
CONST VAR PROCEDURE CALL BEGIN END IF THEN WHILE DO
  = # < <= > >= , ; ( ) := + - * / ident number .
```

Além disso, a linguagem foi simplificada para reduzir a necessidade de submáquinas. Algumas derivações foram reunidas em uma só, o resultado é o que segue:

```
program = block "." .

block = [ "CONST" ident "=" number {"," ident "=" number} ";"]
        [ "VAR" ident {"," ident} ";"]
        { "PROCEDURE" ident ";" block ";" } statement .

statement = [ident ":=" expression |
            "CALL" ident |
            "OUT" expression |
            "IN" ident |
            "BEGIN" statement {";" statement } "END" |
            "IF" expression ("="|"#"|"<"|"<="|">"|">=") expression "THEN" statement |
            "WHILE" expression ("="|"#"|"<"|"<="|">"|">=") expression "DO" statement].

expression = ["+"|"-"]
             ( (ident | number | "(" expression ")")
               { ("*"|"/") (ident | number | "(" expression ")") } )
             { ("+"|"-") ( (ident | number | "(" expression ")")
                           { ("*"|"/") (ident | number | "(" expression ")") } ) } .
```

Para o restante do projeto esta é a gramática adotada, que pode expressar a necessidade de quatro máquinas para o analisador sintático.

1.3. Análise Léxica

Como a construção de autômatos já iria ser necessária na próxima etapa do projeto (análise sintática), decidimos por não fazer o analisador léxico baseado em autômatos. O analisador léxico (no projeto chamado de "*lexer*") projetado usa expressões regulares (recurso presente na grande maioria das linguagens de programação mais populares) para classificar e identificar cada um dos tokens.

O funcionamento do analisador léxico é simples. O arquivo fonte vai sendo lido caracter por caracter e os caracteres vão sendo aglutinados para formarem palavras. Para cada caracter lido, a palavra formada é testada por um conjunto de expressões regulares e classificada de acordo com a expressão regular que a reconhecer. Isso é repetido até que a palavra resultante não possa mais ser reconhecida por nenhuma das expressões regulares e neste caso o último caracter é retirado da palavra (para entrar no próximo token a ser formado) e a última classificação é a atribuida ao token resultante. Abaixo segue o algoritmo representado em pseudo-linguagem:

```
1. while(true)
2.   carac = read_char(file)
3.   palavra = palavra + carac
4.   for expressao in expressoes
5.     break if palavra.match(expressao)
6.   end
7.   if expressao.nil?
8.     # nenhuma expressão regular bateu
9.     take_back(carac) # devolve o caracter lido
10.    palavra.remove_last_char
11.    # retorna o token com a classificacao adequada para a
12.    # ultima expressao que bateu
13.    return (palavra, encontrada.type)
14.  end
15.  encontrada = expressao
16. end
```

O conjunto de expressões regulares usado, e a classificação do token correspondente a cada uma delas está listado a seguir. As expressões regulares na linguagem em que o compilador foi escrito (Ruby) são delimitadas por '/' e alguns caracteres tem sentido especial dentro de uma expressão regular, como o '^' que significa começo de texto e o '$' que significa fim de texto. Para fazer com que os caracteres especiais sejam tratados apenas como texto, devem ser precedidos pelo caracter de escape: '\'.

```
Exp. Regular  => Tipo do Token
/^\.$/        => '.'
/^CONST$/     => 'const'
/^VAR$/       => 'var'
/^PROCEDURE$/ => 'procedure'
/^CALL$/      => 'call'
/^OUT$/       => 'out'
/^IN$/        => 'in'
/^BEGIN$/     => 'begin'
/^END$/       => 'end'
/^IF$/        => 'if'
/^THEN$/      => 'then'
/^WHILE$/     => 'while'
/^DO$/        => 'do'
/^\=$/        => '='
/^\#$/        => '#'
/^<=$/        => '<='
/^<$/         => '<'
/^>=$/        => '>='
```

```
/^>$/          => '>'
/^,$/          => ','
/^;$/          => ';'
/^\($/         => '('
/^\)$/         => ')'
/^:=$/         => ':='
/^\+$/         => '+'
/^-$/          => '-'
/^\/$/         => '/'
/^\*$/         => '*'
/^\d+$/        => 'number'
/^[a-z]\w*$/   => 'identifier'
```

De fato, as expressões regulares são sempre implementadas com autômatos, portanto do ponto de vista da abstração mais baixa, o analisador léxico está implementado com autômatos. A diferença é que a cada novo caracter, o autômato é reiniciado desde o primeiro estado e todas as transições são novamente aplicadas.

Temos ciência de que esta não é a forma mais eficiente de implementar o autômato, porém foi uma forma simples que encontramos de implementá-lo no menor tempo possível. Além disso, o resultado final foi bem satisfatório, já que a compilação não se mostrou demorada e os computadores atuais tem um processamento bastante grande que nos permite simplificar o desenvolvimento dos programas ao troco de performance.

De qualquer forma, a próxima etapa exige a implementação de autômatos, portanto este objetivo da disciplina não fica comprometido.

1.4. Construção dos autômatos através da gramática

Com a técnica de construção de autômatos reconhecedores a partir de gramáticas na notação de Wirth vista em aula, desenhamos as quatro máquinas de estado (autômatos de pilha, não determinísticos) referentes a cada uma das quatro regras da gramática adotada (*program*, *block*, *statement*, *aritmetic_expression*).

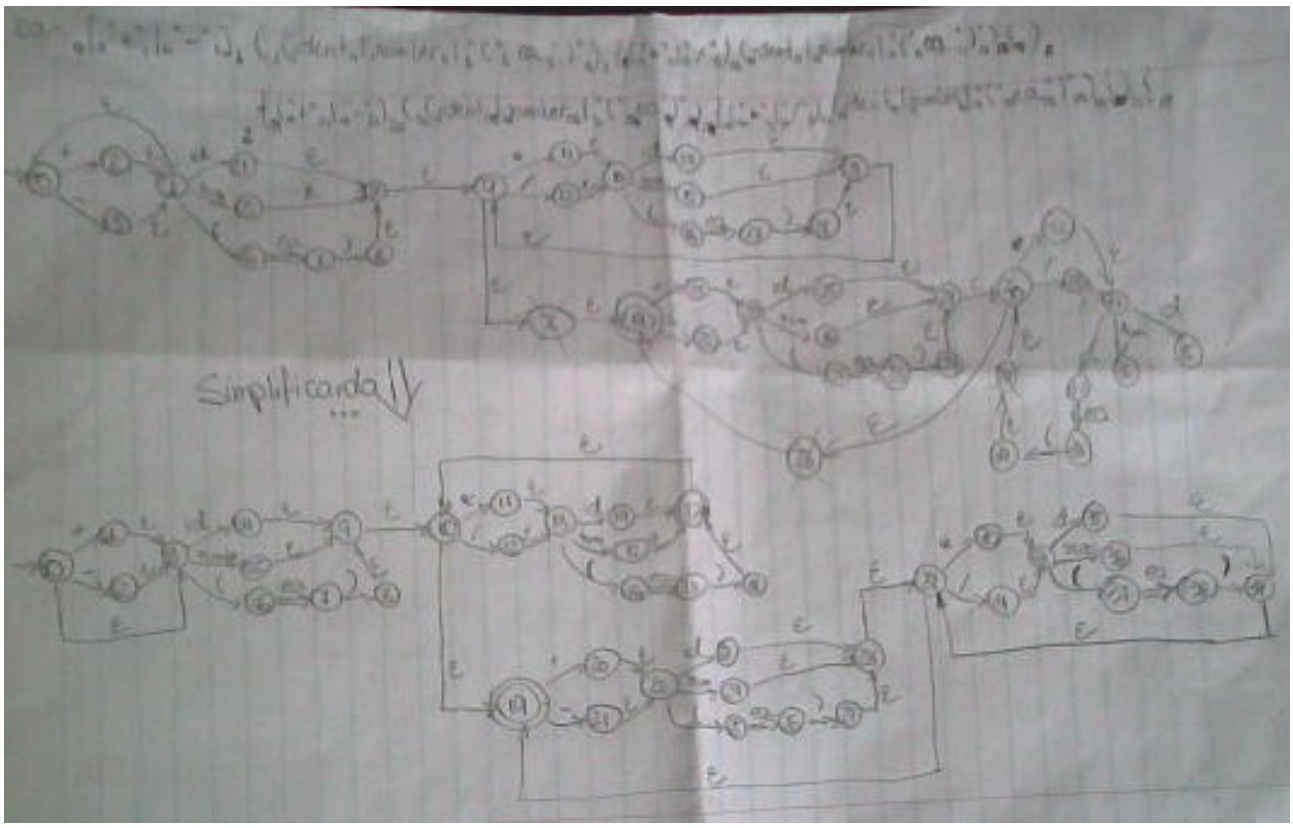

Figura 1 – Autômato responsável pelas expressões aritméticas.

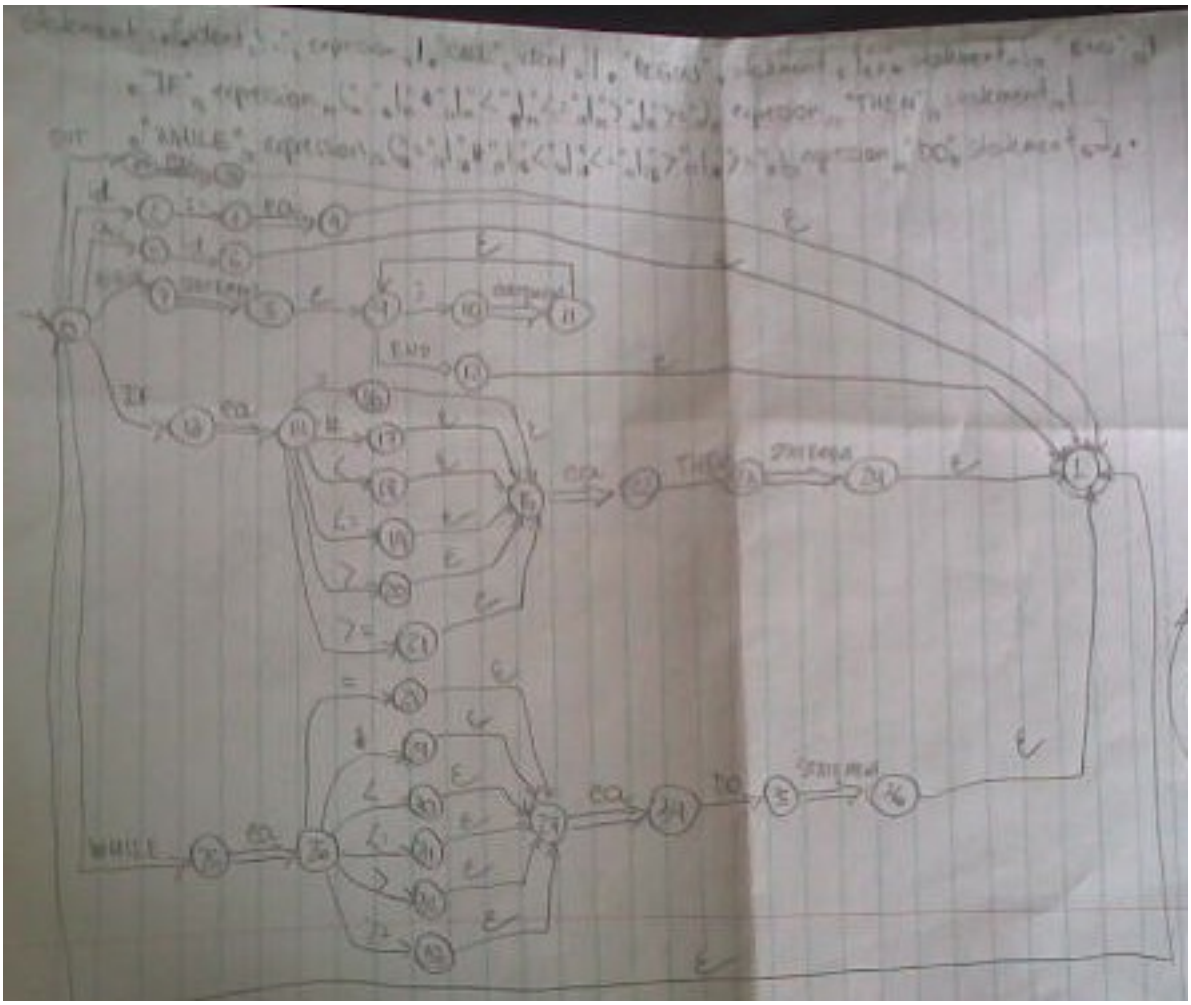

Figura 2 – Autômato responsável pelos *statements*.

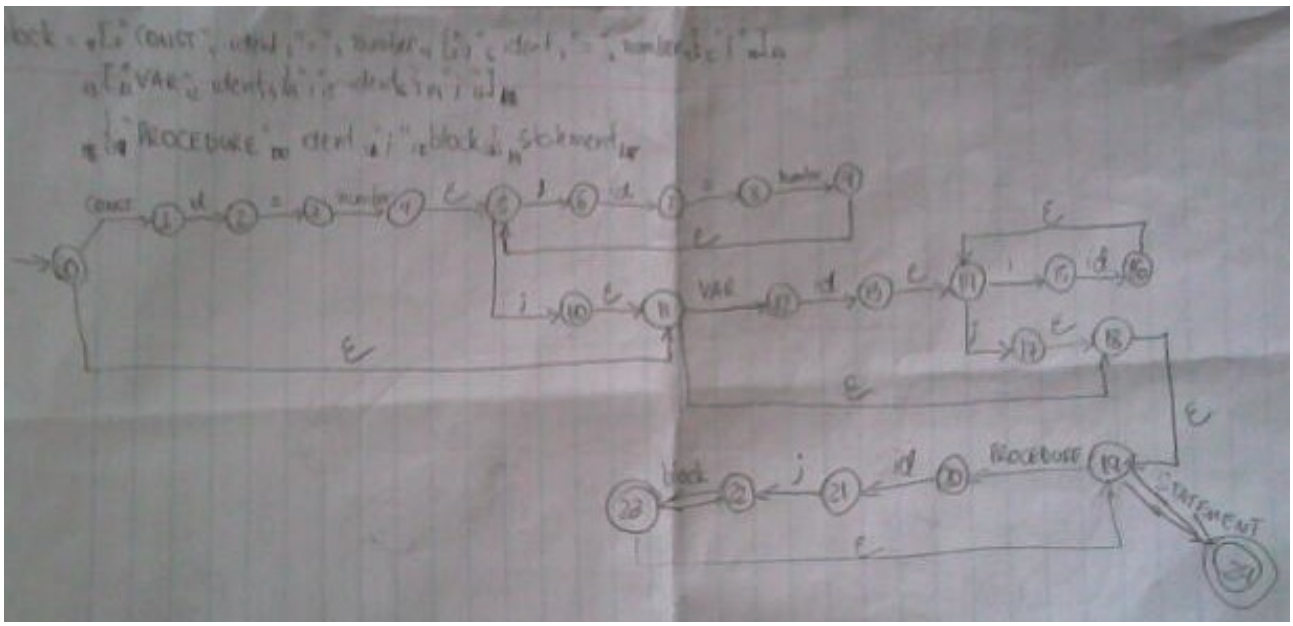

Figura 3 – Autômato responsável pela regra block.

1.5. Implementação dos autômatos

Autômatos costumam ser facilmente implementados com o uso de operações como 'goto' ou grandes encadeamentos de ifs e switches. Tais técnicas não se encaixam muito bem em linguagens orientadas a objetos, onde o uso de gotos costuma ser condenado, bem como o uso de encadeamentos muito grandes de 'ifs' e 'elses', que costumam ser substituídos pelo uso do polimorfismo (uma das características mais interessantes das linguagens orientadas a objeto).

Até o último quadrimestre, desconhecíamos alguma implementação interessante de autômatos em linguagens orientadas a objeto, porém no curso de Linguagens Formais do professor Ricardo Rocha, fomos apresentados a linguagens funcionais (LISP Scheme) e conhecemos uma alternativa interessante para implementação de máquinas de estado / autômatos.

As linguagens funcionais costumam representar cada um dos estados como funções, e as transições são chamadas de funções. Trazendo essa idéia para linguagens orientadas a objeto, cada estado poderia ser representado como um método, e as transições são chamadas de métodos.

Para representar então as quatro máquinas do projeto necessárias, são necessárias quatro classes. Aproveitando a expressividade, flexibilidade e dinamicidade da linguagem Ruby, criamos ainda uma linguagem específica - Domain Specific Language (DSL) – para a definição de autômatos. Segue abaixo um exemplo de como ficam as definições dos autômatos, que ficam fáceis de serem compreendidos.

```
class ExampleMachine
  include StateMachine

  initial_state :s0 do
    transition '.' do
      :s1
    end
    transition '+' do
      :s2
    end
    transition ',' do
      :s3
    end
  end
```

```
  final_state :s1

  state :s2 do
    transition 'identifier' do
      :s0
    end
  end

  final_state :s3 do
    transition '*' do
      :s2
    end
  end
end
```

Qualquer classe pode ser tratada como máquina de estado, bastando para isso incluir o módulo *StateMachine*. Dentro da máquina podem ser definidos quantos estados forem necessários (inclusos o estado inicial e estados finais) e cada estado pode conter quantas transições forem necessárias. Cada transição pode ainda consumir um token, ou ser uma transição vazia (equivalente a consumir o token `ε`) e é responsável por retornar um símbolo que indica qual o próximo estado.

Desta forma as máquinas de estado podem ser representadas declarativamente e não de forma imperativa, o que as torna muito próximas da representação gráfica e facilita mudanças/manutenção. Os autômatos criados na etapa anterior foram transformados em código usando a linguagem específica (DSL), criada para este fim.

Além disso foi criado um conjunto de classes responsável por executar estas máquinas de estado, consumindo tokens fornecidos pelo analisador léxico já apresentado. Tal conjunto de classes pode ser considerado como a *engine* de execução de máquinas de estado.

Os autômatos resultantes da gramática utilizada infelizmente não são determinísticos, porém nenhuma técnica de conversão para autômatos determinísticos foi necessária, tampouco a execução paralela dos pontos de não determinismo. Na prática o não determinismo é facilmente resolvido com uma operação de *lookahead*. Desta forma, se durante a execução algum estado for não determinístico, é feita uma operação de *lookahead* em que o analisador léxico fornece o próximo token a ser lido e assim pode ser determinada a transição adequada sem necessário que todas sejam executadas em paralelo.

É importante enfatizar que isto só foi possível por uma propriedade específica dos autômatos resultantes da gramática na notação de Wirth adotada; apesar de serem não determinísticos, na prática há no máximo uma transição em vazio para cada estado e não há mais de uma transição que consuma o mesmo token. Desta forma, o *lookahead* consegue determinar qual das transições poderá consumir o próximo token, caso nenhuma delas seja adequada, é tomada a transição em vazio.

Neste ponto, tínhamos um completo reconhecedor para a linguagem PL/0 com ligeiras modificações adotada. Diversos testes foram feitos (adotamos uma metodologia de programação guiada por testes – Test Driven Development – TDD) e comprovaram o funcionamento do reconhecedor. Recomendamos uma breve inspeção nos teste unitários construídos; foi usada a biblioteca RSpec para tal, que permite com que os testes sejam uma especificação executável do código e funcionam como documentação de seu funcionamento.

Os testes se encontram na pasta '`test/`' da raiz do projeto e podem ser executados com o comando `rake test` na raiz do projeto caso o interpretador para a linguagem Ruby esteja instalado na máquina.

1.6. Formato dos bytecodes

A próxima etapa vai envolver a geração de código, portanto foi crucial o entendimento dos bytecodes antes da construção das rotinas semânticas.

Já tínhamos conhecimento do funcionamento da linguagem Java e dos seus conceitos de orientação a objetos, por isso o entendimento da estrutura e funcionamento dos bytecodes, bem como sua interpretação pela máquina virtual ficou facilitada.

A especificação da máquina virtual Java define um número relativamente pequeno de opcodes que aceita. Pode-se dizer que os bytecodes são simples. À execução de cada método é associado um *frame*, que guarda um array com os seus argumentos e variáveis locais, além de uma pilha, conhecida como *pilha dos operandos*. A pilha dos operandos é o elemento principal dos frames, já que toda execução dos bytecodes é baseada nela. Os bytecodes sempre operam sobre a pilha de operandos: retiram seus argumentos dela e empilham de volta o resultado.

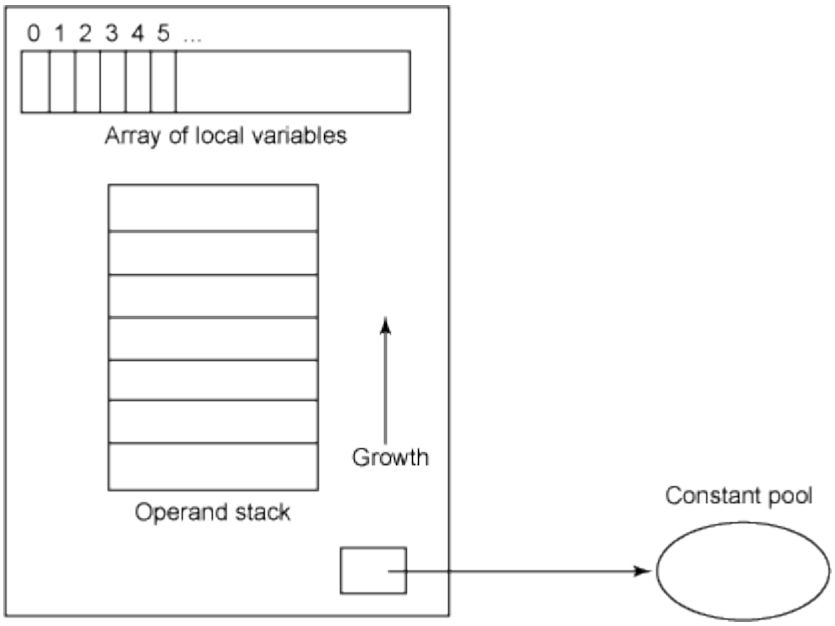

Figura 4 – Representação gráfica de um Frame.

Além disso, os bytecodes provêem todo o suporte à orientação a objetos, inclusive com invocação dinâmica de métodos (*dinamic dispatch*, ou métodos virtuais). Como a linguagem a ser compilada não inclui funcionalidades de orientação a objetos, pouco disto foi utilizado, e a maioria dos artefatos gerados (campos, variáveis e métodos) é estático. O uso de Java com tudo estático é considerado um "uso procedural" da linguagem.

Apesar disso, o entendimento de todos os recursos oferecidos pelos bytecodes foi essencial até para a implementação de rotinas como `IN` e `OUT`, que se beneficiam de recursos de entrada e saída da máquina virtual, e estes estão implementados puramente orientados a objeto.

Gerar os bytecodes puros poderia se tornar uma tarefa difícil, dado que fica difícil de ler o código resultante e entender rapidamente o seu significado. Por este motivo, fomos a procura de algum assembler compatível com os bytecodes Java e encontramos o projeto Jasmin (http://jasmin.sourceforge.net). Jasmin é um montador que recebe como entrada o código na forma de mnemônicos e o transforma em um conjunto executável de bytecodes. Desta forma, o grupo decidiu por fazer a saída do compilador sendo projetado ser código na forma de mnemônicos no formato que o Jasmin espera. O resultado da compilação deve então passar pelo montador Jasmin para que se torne bytecodes executáveis pela máquina virtual Java.

Nós tivemos ainda a oportunidade de indicar o uso do Jasmin para outras equipes que também tinham interesse em aprender um pouco a estrutura dos bytecodes Java. Tivemos notícia de pelo menos mais duas duplas que decidiram usá-lo como alternativa à geração de mnemônicos para o montador e interpretador *MVN*, utilizado no laboratório de fundamentos da computação.

1.7. Semântica

A forma com que as máquinas estão sendo definidas facilitou a inclusão de rotinas semânticas, já que basta adicionar o código necessário dentro de cada uma das transições. A geração de código foi feita na linha do que foi ensinado durante a disciplina. Vale a pena comentar apenas os aspectos mais importantes e as decisões específicas deste projeto.

Um ponto importante a ser mencionado é que independente de como for o código de entrada, apenas uma classe Java é gerada. O corpo principal (statement) do código PL/0 vira o método *main* (`public static void main()`) desta classe, cada um dos procedimentos definidos vira um método estático da classe e cada variável (independente do escopo) vira um atributo estático da classe.

A primeira questão que poderia ser levantada é o uso de métodos estáticos ao invés de métodos de instância. Como a linguagem de entrada não suporta orientação a objetos, não há a necessidade real de instanciar objetos, já que não haverá polimorfismo e invocação dinâmica. Por isso a escolha por métodos estáticos.

Outra questão possível é o uso de atributos estáticos para cada variável do programa em PL/0 ao invés de variáveis locais aos métodos Java. A razão para isso é que as variáveis em PL/0 são compartilhadas entre todos os procedimentos internos ao escopo em que são definidas. Desta forma, foi escolhido o uso de atributos estáticos para que todos os procedimentos (que se tornam métodos) possam ter acesso a estes atributos. A outra opção seria o uso de variáveis locais nos métodos para cada variável PL/0 e neste caso, para os procedimentos definidos internamente (depois) da declaração da variável poderem ter acesso a ela, seria necessária a passagem de todas as variáveis compartilhadas como argumento a todos os procedimentos internos. Com certeza mais trabalhoso.

Como todas as variáveis se tornam atributos estáticos, teoricamente qualquer método/procedimento tem acesso aos atributos. O controle de acesso fica sendo responsabilidade das rotinas semânticas e foi implementado de forma simples. Na transição da máquina de estados que define um novo procedimento, a rotina semântica cria um novo escopo (contexto) filho do contexto atual e que guardará as variáveis definidas dentro deste procedimento. A tabela de símbolos fica sendo na verdade "hierárquica", pois a resolução cada variável é procurada apenas no contexto/escopo atual e nos contextos pais.

a) Expressões Aritméticas

Decidimos seguir a mesma estratégia adotada na construção dos reconhecedores (analisadores sintáticos) e fazer uma abordagem "*bottom-up*". O trabalho foi iniciado pela máquina mais específica (de expressões aritméticas) e foi seguindo até a mais geral e abrangente (*'program'*).

Duas características do projeto facilitaram bastante a implementação das rotinas semânticas de geração de código das expressões aritméticas. O primeiro ponto foi a gramática, que já define a prioridade das operações '`*`' e '`/`' sobre as operações '`+`' e '`-`'. Como a prioridade já está embutida na gramática (e por consequência no autômato resultante), não é necessário fazer o controle de prioridade nas rotinas semânticas com uma pilha de operadores pendentes.

Outro ponto que facilitou bastante a implementação foi o fato de que o bytecode já é baseado em uma pilha; os opcodes sempre operam na pilha de operandos. Um exemplo é a geração de código para a operação '`123 + 456`', que é feita como a seguir:

1. Recebe o token `(123, number)`;
2. empilha 123 na pilha de operandos: `ldc 123`;
3. recebe o token `(+, +)`;
4. guarda `iadd` (opcode para soma) no espaco de variáveis da máquina corrente como '`operação pendente`';
5. recebe o token `(456, number)`;

6. empilha 456 na pilha de operandos: `ldc 456`;
7. gera a linha de código da operação pendente, que espera que os dois operandos já estejam empilhados: `iadd` (soma de dois inteiros).

O único detalhe específico para este projeto é que a operação `iadd` (como quase todas as outras) tira os seus dois operandos da pilha, portanto antes de executá-la, foi necessário gerar o código que empilha os dois operandos (seria equivalente a dizer que as operações aritméticas são feitas em notação polonesa reversa – '`123 456 +`'). Para tal, a máquina de estados tem uma variável que guarda a operação pendente a ser executada (já que a entrada não está na notação polonesa reversa) e esta variável foi implementada como uma variável de instância da classe que representa a máquina de expressões aritméticas.

Outro ponto que foi tratado durante esta etapa é a resolução de variáveis, já que as expressões aritméticas podem envolvê-las. Segue um exemplo para a geração de código de '`a + 1`':

1. Recebe o token (`a, identifier`);
2. procura o símbolo '`a`' na tabela de símbolos (hierárquica) para descobrir qual o atributo correspondente;
3. gera a instrução '`getstatic <nome-do-atributo>`' que é responsável por ler e empilhar o valor corrente do atributo correspondente ao identificador recebido;
4. recebe o token (`+, +`);
5. guarda a operação '`iadd`' como operação pendente;
6. recebe o token (`1, 1`);
7. empilha o número 1 (`ldc 1`);
8. gera a instrução para a operação pendente: `iadd`

O aninhamento de expressões aritméticas é automaticamente tratado pela chamada de submáquina, com a sua própria execução e seus próprios atributos de instância (inclui-se o '`operação pendente`').

b) Statements

O primeiro desafio para as rotinas semânticas desta máquina foi a atribuição. A forma mais simples de expor a implementação é mostrando um exemplo simples. Para a atribuição '`a := 1 + 2`', a máquina:

1. Recebe o token (`a, identifier`);
2. procura na tabela de símbolos o atributo correspondente ao símbolo '`a`';
3. guarda o atributo encontrado na variável (de instância) '`campo_para_atribuicao`';
4. recebe o token (`:=, :=`);
5. chama a submáquina de expressões aritméticas, que deixará o resultado da expressão no topo da pilha;
6. finalmente, gera a instrução '`putstatic campo_para_atribuicao`', que retira o topo da pilha e armazena no atributo estático anteriormente encontrado.

Para a instrução `CALL`, o código gerado é apenas uma chamada do método estático Java correspondente ao procedimento PL/0: `invokestatic <nome-do-método>`.

Statements compostos são tratadas de forma simples, com chamadas de submáquinas, como já feito com as expressões aritméticas.

As rotinas de `IN` e `OUT` se transformam no código Java equivalente a leitura da entrada padrão (teclado) e escrita na saída padrão (monitor).

`OUT <expressao-aritmetica>` deve virar (código Java):

```
Integer valor = Integer.valueOf(<expressao-aritmetica>);
System.out.println(valor.toString());
```

E o equivalente em bytecodes é:

```
getstatic java/lang/System/out Ljava/io/PrintStream;
; ... execução da expressão aritmética (resultado fica no topo da pilha) ...
invokestatic java/lang/Integer/valueOf(I)Ljava/lang/Integer;
invokevirtual java/lang/Object/toString()Ljava/lang/String;
invokevirtual java/io/PrintStream/println(Ljava/lang/String;)V
```

Antes da execução da expressão aritmética, é empilhado o objeto `System.out`, do tipo `java.io.PrintStream`, para ser usado mais a frente.

A primeira instrução transforma o topo da pilha (do tipo '`int`', tipo primitivo resultado da expressão aritmética) em um objeto `Integer` (chamada de `valueOf()`). A segunda transforma o objeto `Integer` em uma `String` com a chamada do método `toString()` e a terceria finalmente é equivalente a chamada de `println(String)`, que retira o argumento da pilha (resultado da expressão, transformado em `String`) e também desempilha o alvo de sua execução (`System.out`, por isto `System.out.println(valor)`). É por este motivo que foi necessário empilhar no começo o objeto `System.out`.

`IN <identificador>` deve virar:

```
BufferedReader reader = getReader();
String readValue = reader.readLine();
Classe.campoCorrespondenteAoIdentificador = Integer.valueOf(readValue);
```

O método `getReader()` foi criado para evitar a criação do objeto BufferedReader toda vez que for necessário fazer uma leitura. O método cria o objeto apenas na primeira chamada e guarda em um atributo (`public static final BufferedReader READER;`). Nas chamadas subsequentes, o método `getReader()` apenas retorna o atributo anteriormente preenchido.

O equivalente em bytecodes fica:

```
invokestatic ClasseGerada/getReader()Ljava/io/BufferedReader;
invokevirtual java/io/BufferedReader/readLine()Ljava/lang/String;
invokestatic java/lang/Integer/valueOf(Ljava/lang/String;)Ljava/lang/Integer;
invokevirtual java/lang/Integer/intValue()I
putstatic ClasseGerada/campoCorrespondenteAoIdentificador I
```

Pouca diferença do código até agora visto. Cabe comentar que alguns opcodes como `putstatic` e `getstatic` recebem como parâmetro o nome completo e o tipo (`I` para `int`) do atributo a ser manipulado. Além disso o campo correspondente ao identificador recebido é descoberto na tabela de símbolos, como antes.

As expressões booleanas são geradas de forma análoga às aritméticas. Exemplo de geração para `IF variavel = 2 * 3 THEN statement`:

1. Recebe token (`IF, IF`);
2. chama submáquina para expressões aritméticas, resultado no topo da pilha;
3. recebe token (`=, =`);
4. guarda a operação `if_icmpeq` como `operacao_pendente`;
5. chama submáquina para expressões aritméticas, resultado no topo da pilha;
6. gera label para o conteudo do bloco `IF`: `#IF_BEGIN`;

7. gera código da `operacao_pendente` (`if_icmpeq #IF_BEGIN`), que desempilha dois inteiros e pula para o label passado como parâmetro se forem iguais;
8. recebe token `(THEN, THEN)`;
9. gera label para o fim do bloco do `IF`: `#IF_END`;
10. gera instrução que sai do `IF` (caso a condição seja falsa): `goto #IF_END;`
11. coloca o label `#IF_BEGIN` na linha atual;
12. chama a submáquina para *Statements*;
13. coloca o label `#IF_END` na linha atual;

O ponto principal para a geração correta do código do `IF` foi a correta geração e posicionamento dos labels. O exemplo foi para a comparação '=', porém o código é o mesmo para todas as outras comparações, trocando-se apenas o opcode gerado para a comparação. As comparações utilizadas foram: `if_icmpeq`, `if_icmpne`, `if_icmplt`, `if_icmple`, `if_icmpgt` e `if_icmpge`.

A linguagem PL/0 original não possui o bloco `ELSE`. Mesmo sendo simples a sua geração, não tivemos tempo para tal, e não acreditamos que fosse agregar nada novo ao projeto do compilador.

A geração para o bloco `WHILE` é análoga, porém é necessário um label a mais para o início do while (antes do teste ser feito) e uma lógica de jumps mais complexa. O template utilizado é o que segue:

```
#BEFORE_TEST:
; codigo da primeira expressao aritmetica, resultado no topo da pilha
; codigo da segunda expressao aritmetica, resultado no topo da pilha
if_icmple #WHILE_BEGIN ; <=, mas poderia ser qualquer outro opcode de comparacao
goto #WHILE_END
#WHILE_BEGIN:
; codigo relativo ao statement
goto #BEFORE_TEST ; reinicia o loop
#WHILE_END:
; fim do loop
```

Outro detalhe interessante de ser citado é que a geração de labels inclui um numero sequencial para evitar a colisão de labels entre blocos `IF` e `WHILE` aninhados.

c) Blocks

A máquina entitulada *Blocks* cuida da definição de constantes, variáveis e procedimentos. O primeiro detalhe do projeto relevante a ser comentado é que para cada constante ou variável criada, o compilador marca a necessidade de criação de um atributo estático na classe a ser gerada e cria uma entrada na tabela de símbolos.

Para haver consistência com a linguagem PL/0, em que os procedimentos tem acesso a variáveis definidas em escopos maiores que o dele, a cada nova declaração de procedimentos, a máquina recebe um novo escopo, filho do atual. A tabela de símbolos é hierárquica, sendo que a resolução de um símbolo sempre começa no escopo atual e vai subindo pelos escopos pais, até que o símbolo seja encontrado. Cada escopo guarda a sua tabela de símbolos individual, o escopo pai, os escopos filhos o tamanho da pilha necessário para a execução do código deste escopo (algumas operações, principalmente as que envolvem chamada de submaquinas, aumentam o tamanho necessario da pilha), o tamanho necessário para o array de variaveis locais e o código gerado para este escopo.

No fim da compilação, o compilador inclui na classe um atributo estático para cada entrada na tabela de símbolos, e cria um método para cada 'escopo' criado durante a execução da máquina *Block*, contendo o código gerado dentro deste escopo. O escopo inicial é colocado dentro do método `main` da classe gerada.

2. Manual de operação do compilador

O arquivo `compile.rb`, é o ponto de entrada para a compilação; é quem inicia todo o processo. A pasta `lib/` contém o código fonte de todas as classes desenvolvidas e o diretório `test/` possui os testes unitários, com o auxílio do RSpec.

Requisitos *mínimos*:

- Java Runtime Environment (**JRE**) >= 1.5 instalado (http://java.sun.com);
- comando `java` disponível no `PATH`.

Requisitos *recomendados*:

- Java Development Kit (**JDK**) >= 1.5 instalado (http://java.sun.com);
- interpretador **Ruby** >= 1.8.5 instalado (http://www.ruby-lang.org);
- consequentemente deverão estar disponíveis no `PATH` os comandos `java`, `ruby` e `rake`;
- **RubyGems** (http://www.rubygems.org/read/chapter/3) instalado, consequentemente o comando `gem` deve estar disponível.

Primeira opção, forma simplificada **apenas com o Java instalado**:

- Para este caso, embutimos no compilador um interpretador Ruby feito em Java, o **JRuby**. Neste caso, o código Ruby é interpretado pela própria máquina virtual Java;
- dentro do diretório raiz do compilador, crie um arquivo em formato PL/0. Existem alguns exemplos disponíveis dentro da pasta '`test/fixtures/`', com a extensão `.pl0`;
- para ver as instruções de uso do compilador, execute o arquivo '`compile.bat`' em um terminal (cmd do Windows, ou sh dos unix);
- ainda em um terminal dentro da pasta raiz do projeto, para compilar execute: '`compile <arquivo-entrada> <arquivo-saida>`', por exemplo: '`compile programa.pl0 programa-compilado`';
- neste caso, o compilador gerará dois arquivos: `programa-compilado.j` (mnemônicos no formato Jasmin) e `programa-compilado.class` (bytecode Java executável, montado pelo Jasmin);
- para executar a classe gerada, basta invocar a máquina virtual Java: `java programa-compilado`. Repare que não é necessário adicionar a extensão `.class`.

Segunda opção, com o interpretador **Ruby e Java instalados**:

- Para executar o compilador usando o interpretador Ruby, basta usar o comando: '`ruby compile.rb <arquivo-entrada> <arquivo-saida>`'
- caso se deseje executar todos os testes automáticos do diretório `test/`, basta executar o comando '`rake test`'. Cabe novamente recomendar a leitura dos testes dentro deste diretório, que como indica a metodologia *TDD*, funcionam como descritores/documentação do funcionamento e propósito do código sendo testado;

Caso haja algum problema com os scripts, o comando completo para a execução do compilador pode ser útil:

```
java -cp jruby.jar org.jruby.Main compile.rb <arq-entrada> <arq-saida>
```