Construção de Compiladores

José Romildo Malaquias

25 de maio de 2011

Sumário

1 Representando programas como dados

1.1 Introdução

Os processadores de linguagem sempre fazem alguma manipulação com programas. Normalmente a entrada para o processador é um programa apresentado na forma de uma sequência de caracteres (texto). Esta sequência de caracteres deve ser convertida para uma representação intermediária que reflete a estrutura do programa, tornando o seu processamento mais fácil.

Frequentemente a representação intermediária do programa é feita usando árvores, com vários tipos de nós, cada um com diferentes atributos. Neste capítulo vamos nos familiarizar com esta representação.

1.2 Uma linguagem de programação muito simples

Vamos trabalhar com uma variação da micro linguagem *straight-line*[1] que chamaremos de μLang. Todos os valores são numéricos (inteiros), e a linguagem possui um conjunto reduzido de formas de comandos expressões.

A sintaxe de μLang é mostrada na gramática da tabela 1.1. As regras de produção são apresentadas na primeira coluna. A segunda coluna será discutida na seção 1.3.

Esta gramática é ambígua, uma vez que permite mais de uma representação para um programa. No entanto ela será adotada por ser simples e por não estarmos interessados nos detalhes das análises léxica e sintática. A ambiguidade está nas regras de produção que definem expressões com operadores binários, e pode ser resolvida pela definição de precedência e associatividade dos operadores, como é usual na matemática. Os operadores relacionais '`=`', '`!=`', '`>`', '`<`', '`>=`' e '`<=`' tem precedência menor do que os demais operadores, e não são associativos. Os operadores aritméticos '`*`' e '`/`' tem a maior precedência, enquanto que '`+`' e '`-`' tem precedência intermediária, e todos eles tem associatividade à esquerda.

Um exemplo de programa nesta linguagem é apresentado na listagem 1.1.

Listagem 1.1: *Exemplo de programa em μLang.*

```
begin
    a := 5+3;
    b := { print(a, a-1), 10 * a } ;
    print(b)
end
```

A seguir é apresentada uma semântica informal da linguagem. Cada *Stm* é um comando que pode ser *executado* para produzir algum efeito colateral (mudar o valor de alguma variável ou exibir alguma

[1] A linguagem *straight-line* é descrita por Appel ([1]) na gramática 1.3 no final do capítulo 1.

Tabela 1.1: *Gramática da linguagem de programação μLang.*

Regra de produção			Tipo
Stm	→	**id :=** *Exp*	`AssignStm`
Stm	→	**print (** *ExpList* **)**	`PrintStm`
Stm	→	**begin** *StmList* **end**	`CompoundStm`
StmList	→	*Stm StmListRest*	`List<Stm>`
StmList	→		`List<Stm>`
StmListRest	→	**;** *Stm StmListRest*	`List<Stm>`
StmListRest	→		`List<Stm>`
Exp	→	**id**	`IdExp`
Exp	→	**num**	`NumExp`
Exp	→	**read ( )**	`ReadExp`
Exp	→	*Exp Binop Exp*	`OpExp`
Exp	→	**{** *Stm* **,** *Exp* **}**	`EseqExp`
Exp	→	**(** *Exp* **)**	`Exp`
ExpList	→	*Exp ExpListRest*	`List<Exp>`
ExpList	→		`List<Exp>`
ExpListRest	→	**,** *Exp ExpListRest*	`List<Exp>`
ExpListRest	→		`List<Exp>`
Binop	→	**+**	`OpExp.Op.PLUS`
Binop	→	**-**	`OpExp.Op.MINUS`
Binop	→	*****	`OpExp.Op.TIMES`
Binop	→	**/**	`OpExp.Op.DIV`
Binop	→	**=**	`OpExp.Op.EQ`
Binop	→	**!=**	`OpExp.Op.NE`
Binop	→	**>**	`OpExp.Op.GT`
Binop	→	**<**	`OpExp.Op.LT`
Binop	→	**>=**	`OpExp.Op.GE`
Binop	→	**<=**	`OpExp.Op.LE`

informação), e cada *Exp* é uma expressão que pode ser *avaliada* para produzir um valor, e possivelmente algum efeito colateral. Um programa é um comando.

Comando de atribuição

i := *e*

Avalia-se a expressão *e*, armazenando o seu resultado na variável *i*.

Comando de impressão

print(*e*$_1$, ..., *e*$_n$)

Avaliam-se as expressões $e_1, \ldots, e_n$ da esquerda para a direita, exibindo os resultados, separando-os por espaço e terminando com uma mudança de linha.

Comando composto

begin *s*$_1$; ...; *s*$_n$ end

Executam-se os comandos $s_1, \ldots, s_n$ em sequência.

Expressão identificador

i

O valor da expressão é o conteúdo armazenado na variável *i*.

Expressão constante

num

O valor da expressão é a própria constante numérica *num*.

Expressão de leitura

`read()`

A avaliação da expressão de leitura é feita pela leitura de um número inteiro da entrada padrão. O número lido é o resultado da expressão.

Expressão operação

e_1 *op* e_2

Avalia-se e_1, e então e_2, e o valor da expressão é dado pela aplicação da operação *op* aos resultados. *op* é um dos operadores aritméticos `+`, `-`, `*` e `/`, ou um dos operadores relacionais `=`, `!=`, `>`, `<`, `>=` e `<=`. No resultado de uma operação relacional, *falso* é indicado pelo valor 0, e *verdadeiro* é indicado pelo valor 1.

Expressão sequência

`{` *s*, *e* `}`

Executa-se o comando *s*, para produção de algum efeito colateral, e em seguida avalia-se a expressão *e*, para produzir o resultado final.

O pograma da listagem 1.1 produz a seguinte saída quando executado:

```
8 7
80
```

Exercício 1.1. Determine a saída produzida pelo programa da listagem 1.2, quando executado.

Listagem 1.2: *Outro exemplo de programa em µLang.*

```
begin
    lado := 4;
    area := lado*lado;
    print(lado, area);
    volume := { lado := lado + 1, lado*lado*lado };
    print(lado,volume)
end
```

1.3 Estruturas de dados

A estrutura de dados mais conveniente para representar o programa é uma árvore, com um nó para cada comando e expressão. Cada categoria sintática (correspondente a um símbolo não terminal) é representada por um tipo de nó específico. Assim cada símbolo não terminal corresponde a uma classe. Se há mais de uma forma para a construção representada pelo símbolo não terminal, esta classe será abstrata, com

subclasses concretas para representar cada forma específica da construção. Para cada regra de produção há um construtor na classe (ou alguma subclasse) correspondente ao símbolo não terminal no lado esquerdo da regra.

As categorias sintáticas para a linguagem μLang serão representadas como segue.

- Para representar comandos (classe sintática *Stm*) será utilizada a classe abstrata `Stm`. Assim todo e qualquer comando será representado por uma instância de `Stm`. Para cada forma específica de comando, definida por uma regra de produção específica, será utilizada uma subclasse de `Stm`.
- Para representar expressões (classe sintática *Exp*) será utilizada a classe abstrata `Exp`. Toda e qualquer expressão será uma instância de `Exp`. Para cada forma específica de expressão será utilizada uma subclasse de `Exp`, de acordo com a regra de produção correspondente.
- Listas de comandos e de expressões serão representados pelas classes `List<Stm>` e `List<Exp>`, respectivamente, utilizando a classe genérica `List<T>` do pacote `java.util` de Java.
- Operadores binários serão representados por instâncias da enumeração `OpExp.Op`.

A tabela 1.2 sintetiza as classes usadas na representação de programas em μLang. Observe também que a gramática apresentada na tabela 1.1 está anotada com os tipos que serão utilizados para representar as construções da linguagem.

Tabela 1.2: *Classes utilizadas para representar os programa na linguagem μLang.*

Símbolo não terminal	Classes	
Stm	`Stm`	
	• `AssignStm`	Comando de atribuição
	• `PrintStm`	Comando de impressão
	• `CompoundStm`	Comando composto
Exp	`Exp`	
	• `NumExp`	Expressão constante
	• `IdExp`	Expressão identificador
	• `ReadExp`	Expressão de leitura
	• `OpExp`	Expressão operação binária
	• `EseqExp`	Expressão sequência
StmList	`List<Stm>`	
StmListRest	`List<Stm>`	
ExpList	`List<Exp>`	
ExpListRest	`List<Exp>`	

Os componentes que aparecem no lado direito de cada regra de produção e carregam alguma informação são representados como campos (variáveis de instância) na classe correspondente. Por exemplo, a classe `AssignStm` deverá ter um campo (do tipo `String`) para representar o identificador e outro campo (do tipo `Exp`) para representar a expressão da atribuição.

A figura 1.1 mostra graficamente a árvore que representa o programa da listagem 1.1.

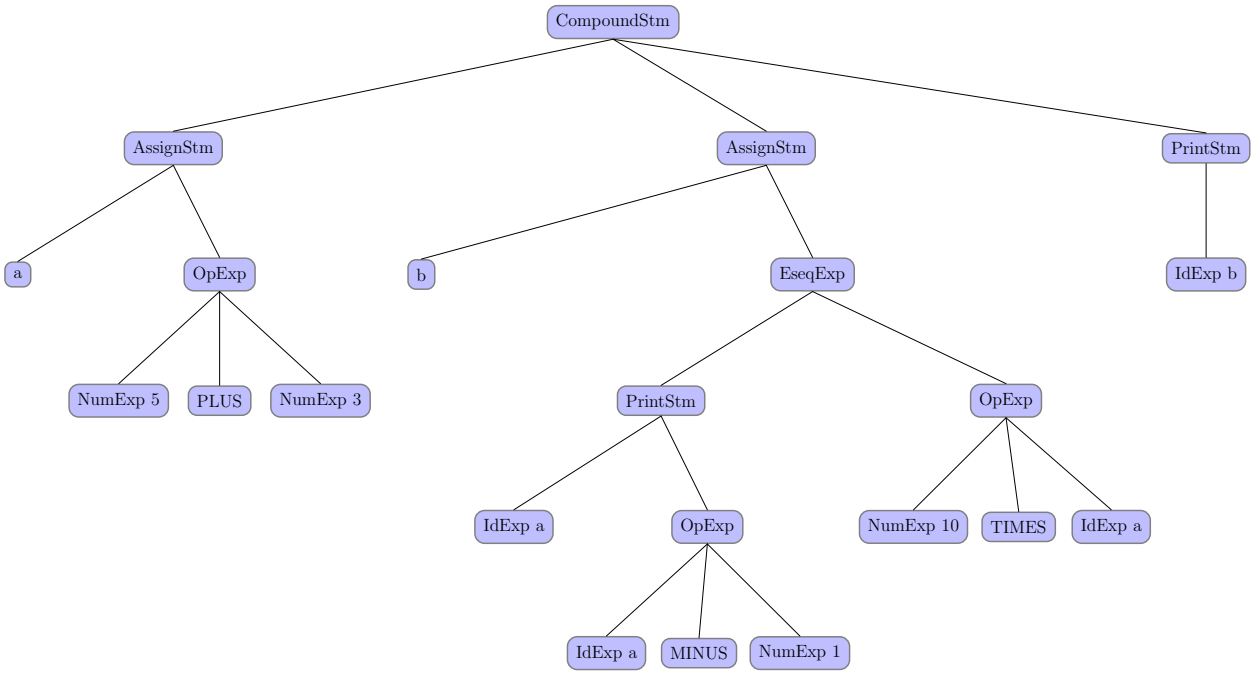
A listagem 1.3 apresenta uma implementação das classes necessárias para representar os programas de μLang.

Listagem 1.3: *Definições das classes para representar programas em μLang.*

```
import java.util.List;

public abstract class Stm
{
```

Figura 1.1: *Representação de árvore para o programa da listagem 1.1.*




```
}

public class AssignStm extends Stm
{
    public String id;
    public Exp exp;

    public AssignStm(String id, Exp exp)
    {
        this.id = id;
        this.exp = exp;
    }
}

public class PrintStm extends Stm
{
    public List<Exp> expList;

    public PrintStm(List<Exp> expList)
    {
        this.expList = expList;
    }
}

public class CompoundStm extends Stm
{
    public List<Stm> stmList;
```

```
    public CompoundStm(List<Stm> stmList)
    {
        this.stmList = stmList;
    }
}

public abstract class Exp
{
}

public class NumExp extends Exp
{
    public Integer num;

    public NumExp(Integer num)
    {
        this.num = num;
    }
}

public class IdExp extends Exp
{
    public String id;

    public IdExp(String id)
    {
        this.id = id;
    }
}

public class ReadExp extends Exp
{
}

public class OpExp extends Exp
{
    public enum Op
    {
        EQ, NE, GT, GE, LT, LE, PLUS, MINUS, TIMES, DIV;
    };

    public Exp left;
    public Op oper;
    public Exp right;

    public OpExp(Exp left, Op oper, Exp right)
    {
        this.left = left;
        this.oper = oper;
        this.right = right;
```

```
    }
}

public class EseqExp extends Exp
{
    public Stm stm;
    public Exp exp;

    public EseqExp(Stm stm, Exp exp)
    {
        this.stm = stm;
        this.exp = exp;
    }
}
```

Exercício 1.2. Represente o programa da listagem 1.4 como uma árvore utilizando as classes apresentadas.

Listagem 1.4: *Programa em µLang.*

```
begin
   k := 2 + 3 * 4 - 1;
   print(k - 1, k, k + 1)
end
```

1.4 Convertendo para árvore de *strings*

Durante o desenvolvimento do processador da linguagem é desejável poder visualizar graficamente a árvore que representa o programa. Para tornar isto possível vamos convertê-la para uma árvore de strings, onde os nós da árvore identificam a classe sintática da estrutura, e as folhas são os componentes da estrutura, como demonstra a figura 1.1. A figura 1.5 exibe a mesma informação formatada no modo texto.

Listagem 1.5: *Visualização da estrutura do programa da listagem 1.1 no modo texto.*

```
CompoundStm
+--AssignStm
|  +--a
|  +--OpExp
|     +--NumExp 5
|     +--PLUS
|     +--NumExp 3
+--AssignStm
|  +--b
|  +--EseqExp
|     +--PrintStm
|     |  +--IdExp a
|     |  +--OpExp
|     |     +--IdExp a
|     |     +--MINUS
|     |     +--NumExp 1
|     +--OpExp
|        +--NumExp 10
|        +--TIMES
|        +--IdExp a
```

```
+--PrintStm
   +--IdExp b
```

A partir da árvore de strings pode-se obter facilmente uma representação visual da árvore de fácil leitura, tanto em modo texto como em modo gráfico.

A classe `Tree` apresentada na listagem 1.6[2] é uma classe genérica que permite fazer esta representação.

- Uma instância de `Tree<E>` é uma árvore contendo uma informação do tipo `E` na raiz, e uma lista de sub-árvores do tipo `Tree<E>`.
- O método `String prettyPrint()` retorna uma *string* que representa a árvore formatada para fácil leitura no modo texto.
- O método `void graph(Formatter out)` insere no objeto de formatação `out` o texto de um programa para a ferramenta *asymptote*[3] representando a árvore formatada para fácil leitura no modo gráfico. Pode-se gravar este programa em *asymptote* em um arquivo texto que, quando compilado, produz uma imagem gráfica da árvore.

Listagem 1.6: *Classe para representação genérica de árvores.*

```
import java.util.Formatter;
import java.util.List;

public class Tree<E>
{
    public E info;
    public List<Tree<E>> children;

    public Tree(E info, List<Tree<E>> children)
    {
      this.info = info;
      this.children = children;
    }

    public String prettyPrint()
    {
      // implementation details omitted
    }

    public void graph(Formatter out)
    {
      // implementation details omitted
    }

    // private methods omitted
}
```

As classes que representam a estrutura do programa (`Stm`, `Exp` e demais) devem ter um método `Tree<String> toTre` que produz uma representação da estrutura como uma árvore de *strings*, permitindo assim a sua fácil visualização. Como exemplo, a listagem 1.7 mostra a definição deste método para a expressão sequência e para o comando de impressão.

[2] A listagem está incompleta. Veja a definição completa desta e de outras classes nos arquivos disponibilizados na página do curso na internet (`http://www.decom.ufop.br/romildo/bcc328.2011-1/`).

[3] Veja `http://www.asymptote.org` para maiores detalhes.

Listagem 1.7: *Conversão de expressão sequência e comando de impressão em árvores de* strings.

```
class EseqExp extends Exp
{
    // instance variables and other methods omitted

    protected Tree<String> toTree()
    {
        List<Tree<String>> l = new LinkedList<Tree<String>>();
        l.add(stm.toTree());
        l.add(exp.toTree());
        return new Tree<String>("EseqExp", l);
    }
}

public class PrintStm extends Stm
{
    // instance variables and other methods omitted

    protected Tree<String> toTree()
    {
        List<Tree<String>> l = new LinkedList<Tree<String>>();
        for (Exp exp : expList)
            l.add(exp.toTree());
        return new Tree<String>("PrintStm", l);
    }
}
```

1.5 Projeto

Exercício 1.3. Utilize os arquivos disponíveis em `http://www.decom.ufop.br/romildo/bcc328.2011-1/praticas/ulang-0.1.zip` para criar uma aplicação em Java para processamento de programas na linguagem μLang. Estão incluídas todas as classes mencionadas até o momento neste capítulo, além de outras classes complementares:

- Classes `Stm`, `AssignStm`, `PrintStm` e `CompoundStm` para representação de comandos.
- Classes `Exp`, `NumExp`, `IdExp`, `ReadExp`, `OpExp` e `EseqExp` para representação de expressões.
- Classe `Tree<E>` para representação de árvores genéricas.
- Classes `Token` e `Lexer` para análise léxica.
- Classe `Parser` para análise sintática.
- Classe `Main`, onde se encontra a definição do método `main`. O método `main` basicamente faz o seguinte:

  1. Determina qual será a programa (entrada) a ser processado. Basicamente verifica-se se o usuário forneceu algum argumento na linha de comando. Em caso afirmativo, este argumento é o nome do arquivo contendo o texto do programa em μLang a ser processado, e este arquivo é aberto para leitura. Caso contrário, a entrada é feita através do dispositivo de entrada padrão, dando

oportunidade ao usuário de digitar o texto do programa. Neste caso a entrada deve ser encerrada com o caracter que marca fim de arquivo (`Control-Z` no Windows e `Control-D` no Linux e Unix).

2. Constrói uma instância de `Lexer`, para fazer análise léxica utilizando a entrada já definida.
3. Constrói uma instância de `Parser`, para fazer análise sintática, utilizando o analisador léxico para obter os símbolos léxicos.
4. Obtém a representação do programa como uma árvore do tipo `Stm`, utilizando o método `parse` do analisador sintátio.
5. Obtém e exibe uma representação da árvore em modo texto usando o método `toTree`.
6. Grava um arquivo contendo o código em *asymptote* capaz de gerar uma imagem gráfica da árvore.

Exercício 1.4. Teste a aplicação com as entradas apresentadas no decorrer do texto.

1.6 Processando programas

Como primeiro exemplo de processamento de programas, vamos acrescentar um método para analisar um determinado programa em μLang, procurando por todas as ocorrências do comando `print` e contando o número de argumentos do mesmo, com o objetivo de determinar o maior número de argumentos com que o comando `print` é usado. Não podemos esquecer que comandos `print` podem ocorrer em expressões, pois a expressão sequência é formada por um comando e uma expressão.

Exercício 1.5. Modifique a aplicação:

1. Acrescente os métodos abstratos `int maxargs()` nas classes `Stm` e `Exp`.
2. Implemente estes métodos em todas as subclasses concretas de `Stm` e `Exp`, de acordo com exposto anteriormente.
3. Modifique o método `Main.main` para também exibir o resultado de `maxargs` sobre o programa sendo processado.
4. Testar o método `maxargs` com os programas dados anteriormente.

1.7 Interpretando programas

Como um exemplo mais útil de processamento de programas vamos implementar um interpretador para a linguagem μLang. O interpretador irá executar os comandos e quando necessário avaliar as expressões que ocorrem nos comandos.

A memória será representada por um mapeamento (da biblioteca padrão de Java) de `String` em `Integer`. A chave do mapeamento será o nome da variável, e o valor associado à chave será o valor da variável.

Exercício 1.6. Modifique a aplicação:

1. Acrescente o método `void interp(Map<String,Integer>)` à classe `Stm` para interpretar (executar) o comando, e o método `Integer eval(Map<String,Integer>)` à classe `Exp`, para avaliar a expressão.
2. Implemente os métodos `interp` e `eval` em todas as subclasses concretas de `Stm` e `Exp`, respectivamente, de acordo com a semântica da linguagem μLang. Lembre-se que um comando é executado para produzir algum efeito colateral (exibir algum valor ou modificar a memória), e uma expressão é avaliada para determinar o seu valor (e possivelmente produzir algum efeito colateral, quando em em sua estrutura houver um comando).

3. Modifique o método `Main.main` para também interpretar o programa de entrada.

4. Testar o interpretador com os programas dados anteriormente no texto.

1.8 Expandindo a linguagem

Agora vamos introduzir novas construções na linguagem μLang.

1.8.1 Comando condicional

Vamo introduzir o comando condicional na linguagem:

$$\textit{Stm} \rightarrow \texttt{if}\ \textit{Exp}\ \texttt{then}\ \textit{Stm}\ \texttt{else}\ \textit{Stm}$$

Quando executado, o comando

```
if e then c1 else c2
```

avalia a expressão e e analisa o resultado. Se o valor de e for *verdadeiro*, executa o comando c_1 e ignora o comando c_2. Caso contrário, executa o comando c_2 e ignora o comando c_1. Um valor é considerado *verdadeiro* se for diferente de zero. Zero é considerado *falso*.

Exercício 1.7. Modifique a aplicação para dar suporte ao comando condicional:

1. Acrescente a classe `IfStm` como uma subclasse concreta de `Stm`, para representar um comando condicional. A classe `IfStm` deverá ter três variáveis de instância, correspondente à condição e às duas alternativas do comando. O construtor da classe deve inicializar estas variáveis de instância.

2. Na classe `Parser` remova o comentário na linha que chama o construtor da classe `IfStm` e comente a linha seguinte, onde é levantada uma exceção.

3. Implemente o método `maxargs` na classe `IfStm`.

4. Implemente o método `interp` na classe `IfStm`.

5. Implemente o método `toTree` na classe `IfStm`.

6. Escreva alguns programas em μLang usando o comando condicional e use-os para testar a aplicação.

1.8.2 Comando de repetição

Vamo introduzir o comando de repetição na linguagem:

$$\textit{Stm} \rightarrow \texttt{while}\ \textit{Exp}\ \texttt{do}\ \textit{Stm}$$

Quando executado, o comando

```
while e do c
```

avalia a expressão e e analisa o resultado. Se o valor de e for *verdadeiro*, executa o comando c e repete a execução do comando de repetição. Caso contrário, a execução do comando de repetição termina. Um valor é considerado *verdadeiro* se for diferente de zero. Zero é considerado *falso*.

Exercício 1.8. Modifique a aplicação para dar suporte ao comando de repetição:

1. Acrescente a classe `WhileStm` como uma subclasse concreta de `Stm`, para representar um comando de repetição. A classe `WhileStm` deverá ter duas variáveis de instância, correspondente à condição e ao corpo da repetição. O construtor da classe deve inicializar estas variáveis de instância.

2. Na classe `Parser` remova o comentário na linha que chama o construtor da classe `WhileStm` e comente a linha seguinte, onde é levantada uma exceção.

3. Implemente o método `maxargs` na classe `WhileStm`.

4. Implemente o método `interp` na classe `WhileStm`.

5. Implemente o método `toTree` na classe `WhileStm`.

6. Escreva alguns programas em μLang usando o comando de repetição e use-os para testar a aplicação.

Exercício 1.9. Escreva um programa em μLang para calcular o fatorial de um número inteiro a ser informado pelo usuário, exibindo o resultado.

2 Análise Léxica

2.1 Introdução

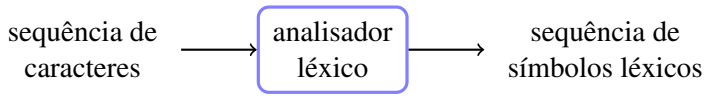
A **análise léxica** é a primeira etapa do processo de compilação e seu objetivo é dividir o código fonte em símbolos, preparado-o para a análise sintática. Neste processo pode-se destacar três atividades como fundamentais:

- extração e classificação dos símbolos léxicos que compõem o programa fonte,
- eliminação de brancos (espaços em branco, tabulação, mudanças de linha) e comentários, e
- recuperação de erros léxicos, gerados por sequências de caracteres que não formam símbolos léxicos.

Símbolos léxicos, ou ***tokens***, são as palavras e sinais de pontuação utilizados para expressar a estrutura de um programa em um texto.

O **analisador léxico**, ou *scanner*, é um módulo do compilador que tem como entrada uma sequência de caracteres (texto do programa), produzindo na saída uma sequência de símbolos léxicos. O analisador léxico atua como uma interface entre o texto de entrada e o analisador sintático.

Figura 2.1: *Analisador léxico.*



As formas mais comuns de símbolos léxicos são:

identificadores palavras utilizadas para nomear entidades do programa, como variáveis, funções, métodos, classes, módulos, etc.

literais sequência de caracteres que representa uma constante, como um número inteiro, um número em ponto flutuante, um caracter, uma *string*, um valor verdade (verdadeiro ou falso), etc.

palavras chaves palavras usados para expressar estruturas da linguagem, como comandos condicionais, comandos de repetição, etc. Geralmente são reservadas, não podendo ser utilizadas como identificadores.

sinais de pontuação sequências de caracteres que auxiliam na construção das estruturas do programa, como por exemplo servindo de separador de expressões em uma lista de expressões.

2.2 Analisador léxico *ad hoc*

Quando a estrutura léxica de uma linguagem não é muito complexa, o analisador léxico pode ser facilmente escrito *à mão*. O programa deve analisar a sequência de caracteres da entrada, agrupando-os para formar os tokens de acordo com a linguagem sendo implementada.

2.3 Especificação de símbolos léxicos

A estrutura léxica das linguagens de programação geralmente são simples o suficiente para a utilização de expressões regulares em sua especificação.

2.3.1 Cadeias e linguagens

Alfabeto é um conjunto finito não vazio de **símbolos**. São exemplos de símbolos letras, dígitos e sinais de pontuação. Exemplos de alfabeto:

- alfabeto binário: $\{0,1\}$
- ASCII
- Unicode

Uma **cadeia** em um alfabeto é uma sequência finita de símbolos desse alfabeto. Por exemplo, 1000101 é uma cadeia no alfabeto binário.

O **tamanho** de uma cadeia s, denotado por $|s|$, é o número de símbolos em s. Por exemplo, $|1000101| = 7$.

A **cadeia vazia**, representada por ε, é a cadeia de tamanho zero.

Um **prefixo** de uma cadeia s é qualquer cadeia obtida pela remoção de zero ou mais símbolos do final de s. Por exemplo, *ama*, *amar*, *a* e ε são prefixos da cadeia *amarelo*.

Um **sufixo** de uma cadeia s é qualquer cadeia obtida pela remoção de zero ou mais símbolos do início de s. Por exemplo, *elo*, *amarelo*, *o* e ε são sufixos da cadeia *amarelo*.

Uma **subcadeia** de uma cadeia s é qualquer cadeia obtida removendo-se qualquer prefixo e qualquer sufixo de s. Por exemplo, *are*, *mar*, *relo* e ε são subcadeias da cadeia *amarelo*.

Um prefixo, um sufixo ou uma subcadeia de uma cadeia s é dito **próprio** se não for a cadeia vazia ε ou a própria cadeia s.

Uma **subsequência** de uma cadeia s é qualquer cadeia formada pela exclusão de zero ou mais símbolos (não necessariamente consecutivos) de s. Por exemplo, *aaeo* é uma subsequência da cadeia *amarelo*.

A **concatenação** xy (também escrita como $x \cdot y$) de duas cadeias x e y é a cadeia formada pelo acréscimo de y ao final de x. Por exemplo, $verde \cdot amarelo = verdeamarelo$. A cadeia vazia é o elemento neutro da concatenação: para qualquer cadeia s, $\varepsilon s = s\varepsilon = s$. A concatenação é associativa: para quaisquer cadeias r, s e t, $(rs)t = r(st) = rst$. Porém a concatenação não é comutativa.

2.3.2 Linguagens

Uma **linguagem** é um conjunto contável de cadeias de algum alfabeto. Exemplos de linguagens:

- o conjunto vazio, ϕ
- o conjunto unitário contendo apenas a cadeia vazia, $\{\varepsilon\}$
- o conjunto dos números binários de 3 dígitos, $\{000, 001, 010, 011, 100, 101, 110, 111\}$
- o conjunto de todos os programas Java sintaticamente bem formados

- o conjunto de todas as sentenças portuguesas gramaticalmente corretas

As operações sobre linguagens mais importantes para a análise léxica são união, concatenação e fechamento.

A união de duas linguagens L e M é dada por

$$L \cup M = \{s | s \in L \text{ ou } s \in M\}$$

A concatenação de duas linguagens L e M é dada por

$$LM = \{st | s \in L \text{ e } t \in M\}$$

Uma potência de uma linguagem L é dada por

$$L^n = \begin{cases} \{\varepsilon\} & \text{se } n = 0 \\ LL^{i-1} & \text{se } n > 0 \end{cases}$$

O fechamento Kleene de uma linguagem L é dado por

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

O fechamento positivo de uma linguagem L é dado por

$$L^+ = \bigcup_{i=1}^{\infty} L^i$$

2.3.3 Expressões regulares

Uma **expressão regular** (também conhecida como **padrão**) descreve um conjunto de cadeias de símbolos de forma concisa, sem precisar listar todos os elementos do conjunto. Por exemplo, o conjunto formado pelas cadeias *Handel*, *Händel* e *Haendel* pode ser descrito pelo padrão `H(ä|ae?)ndel`.

Segundo a teoria das linguagens formais, uma expressão regular sobre um conjunto finito Σ de símbolos (chamado de **alfabeto**) denota uma linguagem regular sobre Σ:

- o **conjunto vazio** ϕ é uma expressão regular que denota a linguagem vazia $\{\}$
- a **cadeia vazia** ε é uma expressão regular que denota a linguagem $\{\varepsilon\}$
- o **literal** $a \in \Sigma$ é uma expressão regular que denota a linguagem $\{a\}$
- se R e S são expressões regulares, então RS, também escrito como $R \cdot S$, é uma expressão regular que denota a linguagem $\{\alpha\beta | \alpha \in L(R) \wedge \beta \in L(S)\}$ cujas cadeias são formadas pela **concatenação** de uma cadeia de $L(R)$ com uma cadeia de $L(S)$
- se R e S são expressões regulares, então $R|S$ é uma expressão regular que denota a linguagem $L(R) \cup L(S)$
- se R é uma expressão regular, então $R\star$ é uma expressão regular que denota a linguagem $\{\varepsilon\} \cup L(R) \cup L(RR) \cup L(RRR) \cup \cdots$

2.4 Geradores de analisadores léxicos

Os geradores de analisadores léxicos são ferramentas que tem como entrada uma especificação da estrutura léxica de uma linguagem (na forma de um arquivo texto), e produzem um analisador léxico correspondente à especificação.

2.5 JFlex

JFLex (`http://jflex.de/`) é um gerador de analisador léxico escrito em Java que gera código em Java. Ele é distribuído usando o licença GPL e está disponível em `http://jflex.de/download.html`. O seu manual pode ser obtido em `http://jflex.de/manual.pdf`.

2.5.1 Instalação

Sendo uma aplicação Java, JFlex necessita de uma máquina virtual Java para ser executado. Assim certifique-se primeiro de que uma máquina virtual de Java já está instalada. Assumiremos que a sua versão é 1.2 ou superior.

Instalação no Windows

Para instalar JFlex no Windows, siga os passos seguintes:

1. Descompacte o arquivo *zip* disponibilizado em `http://jflex.de/download.html` em um diretório de sua preferência. Vamos assumir que este diretório é `C:\`, e que a versão do JFlex é a última disponível (1.4.3, neste momento). Assim o JFlex estará disponível em `C:\jflex-1.4.3`.

2. Edite o arquivo `bin\jflex.bat` de forma que ele fique com o seguinte conteúdo:

   ```
   set JFLEX_HOME=C:\jflex-1.4.3
   java -Xmx128m -jar %JFLEX_HOME%\lib\JFlex.jar %1 %2 %3 %4 %5 %6 %7 %8 %9
   ```

   Se necessáro, modifique a variável de ambiente `JFLEX_HOME` de acordo com o diretório onde o JFlex foi instalado.

3. Inclua o diretório `bin\` do JFlex (no exemplo, `C:\jflex-1.4.3\bin`) na variável de ambiente `PATH`.

Instalação no Linux

Sugiro que se utilize o programa de gerenciamento de pacotes de sua distribuição Linux para instalar o JFlex. Caso não haja um pacote disponível para a sua distribuição, siga as instruções em `http://jflex.de/installing.html`.

Instalação no Eclipse

O JFlex pode ser integrado no Eclipse, um ambiente de desenvolvimento integrado largamente utilizado para o desenvolvimento de aplicações Java. Para tanto siga os passos segintes. Assumiremos que a versão do Eclipse é a versão 3.5.2.

1. Abra a janela de configuração de ferramentas externas acessando o menu *Run -> External Tools -> External Tools Configurations...*

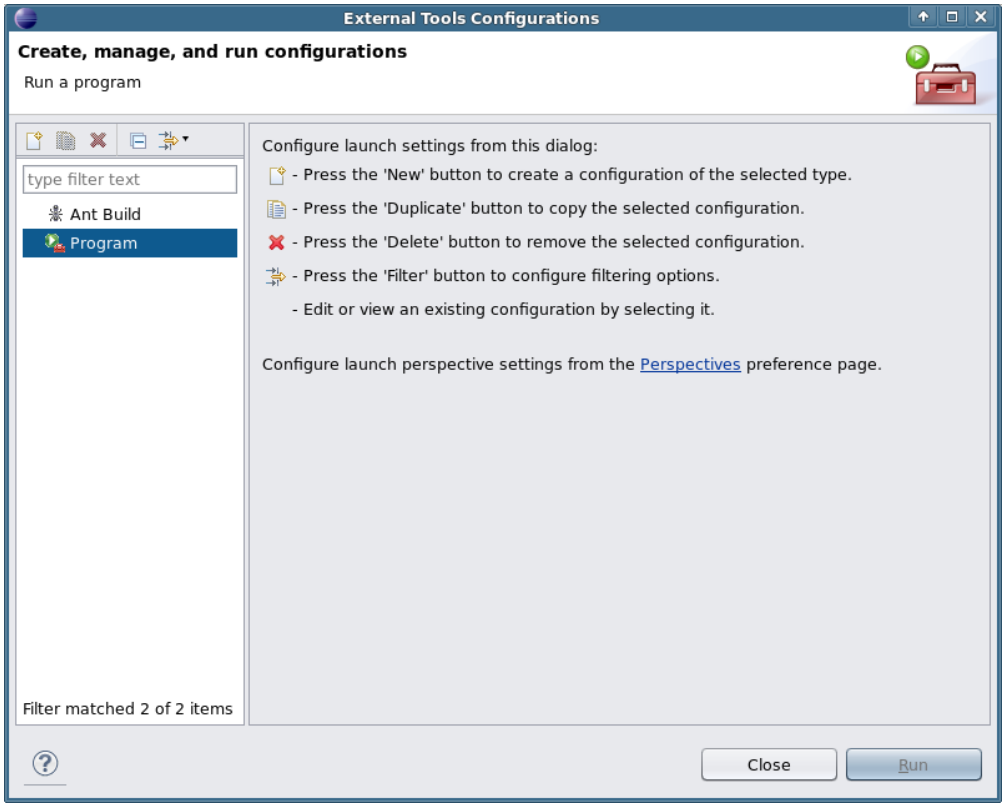
2. Na janela *External Tools Configuration* Clique no botão *New Launch Configuration* para criar uma nova configuração. Veja a figura 2.2.

3. Modifique o nome da configuração para `JFlex`.

4. Na aba *Main* preencha os seguintes campos (veja a figura 2.3):

   - Coloque o caminho para o arquivo de execução do JFlex no campo *Location*. No exemplo do Windows o caminho é `C:\jflex-1.4.3\bin\jflex.bat`. No linux o caminho provavelmente seja `/usr/bin/jflex`.

Figura 2.2: *Criando uma nova configuração de ferramenta externa.*



- Em *Working Directory* coloque `${container_loc}`. A variável `${container_loc}` corresponde ao caminho absoluto no sistema de arquivos do pai (um diretório ou um projeto) do recurso selecionado. No caso o recurso selecionado é o arquivo de especificação do JFlex a ser compilado.
- Em *Arguments* coloque `${resource_name}`. A variável `${resource_name}` corresponde ao nome do recurso selecionado.

5. Na aba *Refresh* marque as opções seguintes (veja a figura 2.4):
   - Marque a opção *Refresh resources upon completion.*
   - Marque a opção *The project containing the selected resource.*
6. Clique no botão *Apply* e depois no botão *Close*.

2.5.2 Executando o JFlex

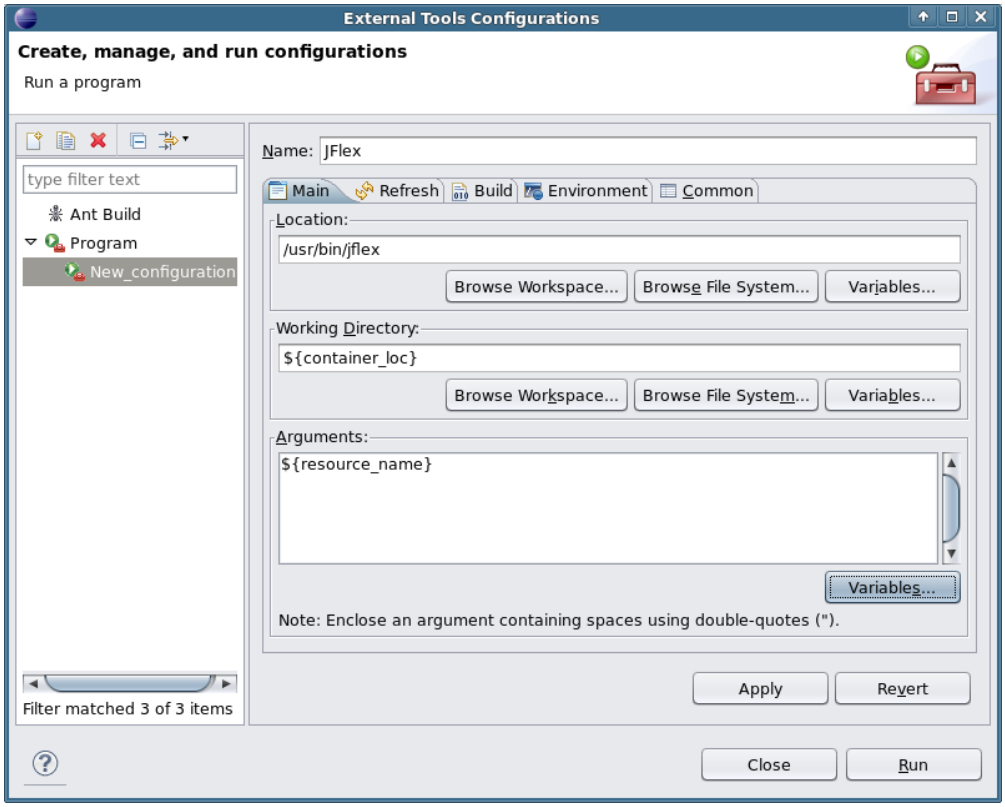
Linha de comando

O JFlex pode ser executado na linha de comando com:

```
jflex <opções> <arquivos de entrada>
```

Consulte o manual para conhecer as opções que o JFlex aceita.

Figura 2.3: *Aba Main da configuração do JFlex no Eclipse.*



No Eclipse

Para compilar uma especificação, abra o arquivo contendo a especificação, vá à janela de configuração de ferramentas externas acessando o menu *Run -> External Tools -> External Tools Configurations...*, selecione a ferramenta JFlex, e clique no botão *Run*. Da próxima vez não será mais necessário abrir esta janela, pois uma entrada do JFlex é adicionada ao menu: *Run -> External Tools -> JFlex*.

Exercício 2.1. Verifique a configuração do JFlex e do Eclipse em seu computador.

2.5.3 Exemplo: analisador simples

Uma especificação léxica é formada por três seções, separadas por `%%`:

1. **Código do usuário**. É inserido no início do arquivo gerado pelo JFlex. Tipicamente colocamos declaração de pacote e declaração de importação nesta seção.

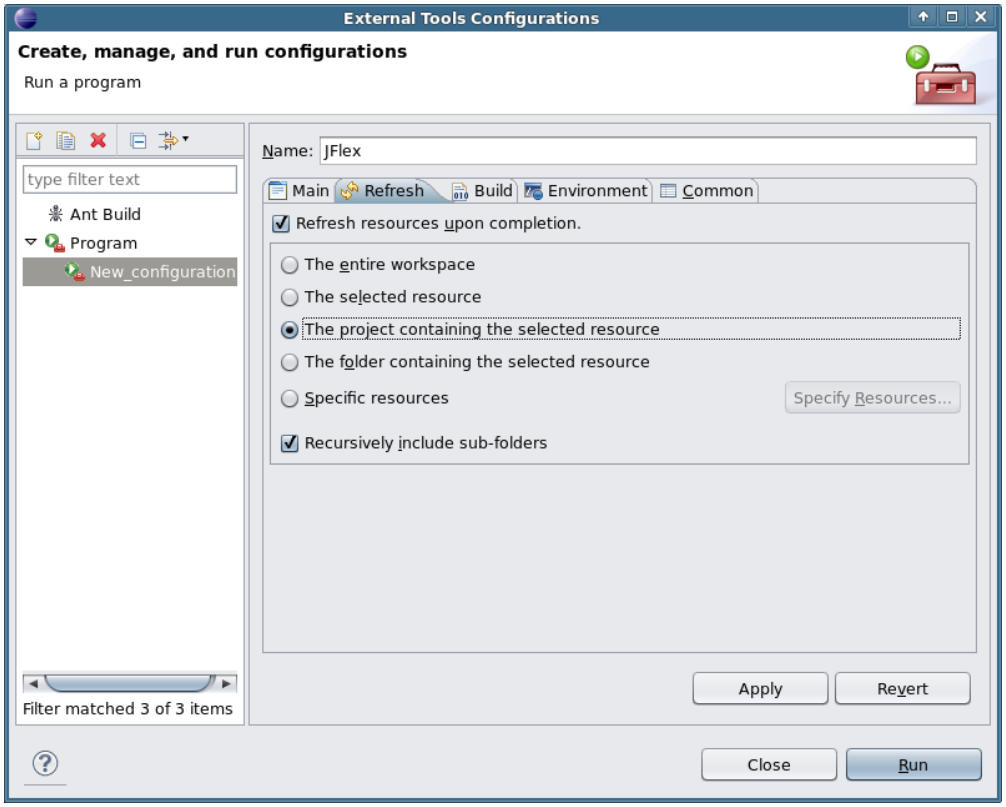
2. **Opções e declarações**. É um conjunto de

   - opções que permitem configurar como o analisador léxico será gerado,
   - declarações de estados, e
   - declarações de macros

   Cada opção deve começar com o caracter `%`, colocado na primeira coluna da linha.

3. **Regras léxicas**. É um conjunto de regras, onde cada regra é formada por uma expressão regular e uma açãoo. A expressão regular identifica uma classe de símbolos léxicos, e a ação é um trecho de código Java que é executado quando um símbolo é formado durante a análise léxica utilizando

Figura 2.4: *Aba Refresh da configuração do JFlex no Eclipse.*



a expressão regular correspondente. Caso haja conflito entre as regras, decidie-se pela regra que produzir a maior cadeia. Se ainda assim persistir o conflito, dá-se preferência à regra que aparece primeiro na especificação. O texto da entrada que casa com a expressão regular pode ser acessado pelo método `yytext`, que é incluído na classe gerada.

A listagem 2.1 contém uma especificação léxica bem simples.

Listagem 2.1: *Exemplo de especificação léxica.*

```
%%

%integer

%%

[a-z][a-z0-9]*  { return 1; }
[0-9]+          { return 2; }
[ \t\n\r]+      { /* do nothing */ }
.               { System.err.printf("error: unexpected char |%s|\n", yytext()); }
```

1. Identificadores são formados por uma sequência de letras minúsculas e dígitos decimais, começando por uma letra. O tipo do símbolo `1`.

2. Números naturais são formados por uma sequência de um ou mais dígitos decimais. O tipo do símbolo é `2`.

3. Brancos (espaços, tabulação horizontal, mudança de linha) são ignorados.

4. Qualquer caracter da entrada que não formar símbolo léxico (identificador ou número inteiro) ou for ignorado (brancos) gera uma mensagem de erro. É importante que esta seja a última regra, pois ela casa com qualquer caracter da entrada (exceto mudança de linha).

Por default, JFlex gera uma classe chamada `Yylex`. A opção `%class` permite especificar o nome da classe desejada. O construtor desta classe tem um argumento representando a entrada a ser analisada.

A classe gerada tem um método público, sem argumentos, que deve ser utilizado no restante da aplicação para obter o próximo símbolo léxico da entrada. O nome default deste método é `yylex`, mas pode-se especificar um outro nome utilizando a opção `%function`. O tipo de retorno deste método de análise léxica é, por default, `YYToken`, que deve ser definido na aplicação. A opção `%type` permite usar um outro tipo, no entanto. Existe também a opção `%integer` que específica o tipo de retorno **`int`**.

Quando o final de arquivo é atingido, o método de análise léxica retorna um valor específico para indicar o fim da entrada. Quando o tipo de retorno for uma subclasse de `java.lang.Object`, o valor retornado é **`null`**, por default. Com `%integer` o valor de retorno default é a constante `YYEOF`, definida como uma variável de instância **`public static final int`** na classe gerada. Existem diferentes maneiras de se especificar um valor de retorno diferente do default quando se atinge o fim de arquivo.

A listagem 2.2 define a classe `Test` com o método `main` que pode ser utilizado para testar o analisador gerado.

Listagem 2.2: *Classe para testar o analisador gerado pelo JFlex.*

```
import java.io.FileReader;
import java.io.Reader;

public class Test
{
    public static void main(String[] args)
    {
        if (args.length != 1)
        {
            System.err.println("usage: java Test <input file>");
            System.exit(1);
        }
        try
        {
            Reader input = new FileReader(args[0]);
            Yylex scanner = new Yylex(input);
            int token;
            do
            {
                token = scanner.yylex();
                System.out.println(token);
            }
            while (token != Yylex.YYEOF);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Basicamente toma-se as seguintes ações:

1. O arquivo especificado na linha de comando é aberto para leitura
2. Uma instância da classe de análise léxica é criada, usando este arquivo como fonte
3. Em um comando de repetição, cada símbolo léxico é obtido e exibido na saída padrão, até que se chegue no final da entrada.

Exercício 2.2. Cria um aplicativo Java para testar o JFlex utilizando a especificação e a classe `Test`, dadas anteriormente. Crie um programa de entrada e teste a aplicação determinando a saída para esta entrada.

2.5.4 Exemplo: usando uma classe para representar os símbols léxicos

A classe `Token`, definida na listagem 2.3, será utilizada para representar os símbolos léxicos.

Listagem 2.3: *Classe para representar um símbolo léxico.*

```
import java.util.Formatter;

public class Token
{
    public enum T
    {
        IF    { public String toString() { return "IF"; } },
        ID    { public String toString() { return "ID"; } },
        INT   { public String toString() { return "INT"; } },
        FLOAT { public String toString() { return "FLOAT"; } },
        STR   { public String toString() { return "STR"; } },
        EOF   { public String toString() { return "EOF"; } }
    }

    public T type;
    public Object val;
    public int line;
    public int col;

    public Token(T type, int line, int col)
    {
        this.type = type;
        this.line = line;
        this.col = col;
    }

    public Token(T type, Object val, int line, int col)
    {
        this.type = type;
        this.val = val;
        this.line = line;
        this.col = col;
    }

    public String toString()
```

```
    {
        Formatter out = new Formatter();
        out.format("(%4d,%4d) %s", line, col, type);
        if (val != null)
            out.format(" [%s]", val);
        return out.toString();
    }
}
```

Cada símbolo léxico tem as seguintes informações:

- o tipo do símbolo léxico, representando a sua categoria, como por exemplo identificador, literal inteiro, literal em ponto flutuante, etc.
- valor semântico, que é alguma informação adicional sobre o símbolo léxico e que será necessária em etapas posteriores da compilação; exemplo: nome do identificador, valor de um literal inteiro, etc.
- posição, dada pelo número da linha e da coluna, em que o símbolo aparece na entrada.

Os tipos possíveis para um símbolo léxico são representados pelos valores enumerados do tipo `Token.T`.

A listagem 2.4 mostra uma especificação léxica simples.

Listagem 2.4: *Exemplo de especificação léxica.*

```
%%

%class Lexer
%type Token
%line
%column

%{
    private Token token(Token.T type)
    {
        return new Token(type, yyline, yycolumn);
    }

    private Token token(Token.T type, Object val)
    {
        return new Token(type, val, yyline, yycolumn);
    }
%}

%%

if              { return token(Token.T.IF); }
[a-z][a-z0-9]*  { return token(Token.T.ID, yytext()); }
[0-9]+          { return token(Token.T.INT, new Integer(yytext())); }
[0-9]+"."[0-9]*|[0-9]*"."[0-9]+
                { return token(Token.T.FLOAT, new Double(yytext())); }
[ \t\n\r]+      { /* do nothing */ }
<<EOF>>         { return token(Token.T.EOF); }
.               { System.err.printf("error: unexpected char |%s|\n", yytext()); }
```

Observe que:

- O nome da classe gerada é `Lexer`.
- O tipo do resultado do método que faz a análise léxica é `Token`.
- As opções `%line`, `%column` e `%char` habilitam a contagem de linhas, colunas e caracteres durante a análise léxica. As variáveis de instância `yyline`, `yycolumn` e `yychar` podem ser utilizadas para se obter a contagem de linha, de coluna e de caracter atual.
- As opções `%{` e `%}` delimitam um código que é inserido na classe gerado. Neste exemplo definimos dois métodos adicionais na classe gerada que facilitam a construção dos símbolos léxicos.
- A expressão regular `<<EOF>>` é utilizada quando o fim da entrada é atingido.

Exercício 2.3. Explique porque a regra da palavra-chave `if` precisa ser colocada antes da regra de identificadores.

Exercício 2.4. Utilize esta especificação léxica para gerar um analisador usando o JFlex. Crie uma nova classe `Test2` baseada na classe `Test` para testar o novo analisador. Teste o analisador.

2.5.5 Exemplo: definição de expressão regular

A listagem 2.5 mostra uma especificação léxica simples.

Listagem 2.5: *Exemplo de especificação léxica.*

```
%%

%class Lexer
%type Token
%line
%column

%{
    private Token token(Token.T type)
    {
        return new Token(type, yyline, yycolumn);
    }

    private Token token(Token.T type, Object val)
    {
        return new Token(type, val, yyline, yycolumn);
    }
%}

alpha = [a-zA-Z]
dig   = [0-9]
id    = {alpha} ({alpha} | {dig})*
int   = {dig}+
float = {dig}+ "." {dig}* | {dig}* "." {dig}+

%%

if          { return token(Token.T.IF); }
{id}        { return token(Token.T.ID, yytext()); }
```

```
{int}       { return token(Token.T.INT, new Integer(yytext())); }
{float}     { return token(Token.T.FLOAT, new Double(yytext())); }
[ \t\n\r]+  { /* do nothing */ }
<<EOF>>     { return token(Token.T.EOF); }
.           { System.err.printf("error: unexpected char |%s|\n", yytext()); }
```

Observe que segunda seção da especificação define cinco expressões regulares, `{alpha}`, `{dig}`, `{id}`, `{int}` e `{float}`, que são utilizadas nas próprias definições e nas regras léxicas.

Exercício 2.5. Gere um analisador léxico usando esta nova especificação, e teste-o.

2.5.6 Exemplo: estados

A listagem 2.6 mostra uma especificação léxica que define novos estados.

Listagem 2.6: *Exemplo de especificação léxica usando estados.*

```
%%

%class Lexer
%type Token
%line
%column

%{
    private StringBuilder str = new StringBuilder();

    private Token token(Token.T type)
    {
        return new Token(type, yyline, yycolumn);
    }

    private Token token(Token.T type, Object val)
    {
        return new Token(type, val, yyline, yycolumn);
    }
%}

%state STR

alpha = [a-zA-Z]
dig   = [0-9]
id    = {alpha} ({alpha} | {dig})*
int   = {dig}+
float = {dig}+ "." {dig}* | {dig}* "." {dig}+

%%

<YYINITIAL> {
if              { return token(Token.T.IF); }
{id}            { return token(Token.T.ID, yytext()); }
{int}           { return token(Token.T.INT, new Integer(yytext())); }
{float}         { return token(Token.T.FLOAT, new Double(yytext())); }
```

```
\"              { str.setLength(0);
                  yybegin(STR);
                }
[ \t\n\r]+       { /* do nothing */ }
<<EOF>>         { return token(Token.T.EOF); }
}

<STR> \"         { yybegin(YYINITIAL);
                  return token(Token.T.STR, str.toString());
                }
<STR> \\t        { str.append('\t'); }
<STR> \\n        { str.append('\n'); }
<STR> \\\"       { str.append('"'); }
<STR> \\\\       { str.append('\\'); }
<STR> [^\n\r\\]+    { str.append(yytext()); }
<STR> <<EOF>>    { yybegin(YYINITIAL);
                  System.err.println("error: unclosed string literal");
                }

.|\n             { System.err.printf("error: unexpected char |%s|\n", yytext()); }
```

Este analisador utiliza um estado especial para analisar literais caracteres. Observe que:

1. O estado inicial é `YYINITIAL`.
2. Outros estados podem ser declarados na segunda seção usando a opção `%state`.
3. O método `yybegin`, incluído pelo JFlex na classe gerada, permite mudar de estado.
4. Uma regra léxica pode ser prefixada com uma lista de estados, indicando que a regra só é usada se o analisador estiver em um dos etados listados.
5. Quanda a lista de estados é omitida de uma regra, a regra pode ser utilizada em qualquer estado.

Exercício 2.6. Gere um analisador léxico usando esta nova especificação, e teste-o.

2.6 Analisador léxico para a linguagem Tiger

Exercício 2.7. Utilizando o JFLex, implementar um analisador léxico para a linguagem Tiger, definida no apêndice do livro do Appel [1].

3 Análise sintática descendente recursiva

3.1 Introdução

A **análise sintática** (também conhecida pelo termo em inglês **parsing**) é o processo de analisar uma sequência de símbolos léxicos (resultantes da análise léxica) para determinar sua estrutura gramatical segundo uma determinada gramática formal. Essa análise faz parte de um compilador, sendo precedida pela análise léxica e sucedida pela análise semântica, como mostra a figura 3.1.

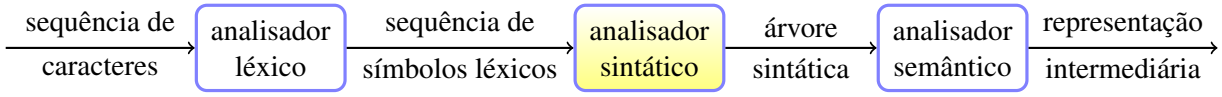


Figura 3.1: *Analisador sintático.*

A análise sintática constrói uma estrutura de dados, em geral uma árvore, chamada de **árvore sintática**, para representar o programa sendo compilado. Esta estrutura de dados captura a hierarquia implícita nas construções que formam o programa, sendo conveniente para o processamento posterior do programa. A figura 3.2 mostra um exemplo.

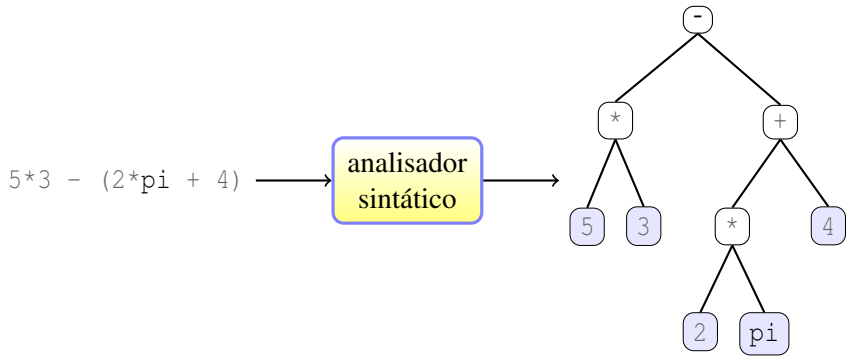


Figura 3.2: *Exemplo da análise sintática de uma expressão aritmética. O resultado é uma árvore da expressão.*

3.2 Gramáticas livres de contexto

A especificação da estrutura sintática das linguagens de programação é feita através de gramáticas livres de contexto.

Uma **gramática livre de contexto** é formada por

- um **conjunto de símbolos terminais**, que são os símbolos básicos a partir dos quais as cadeias são formadas;
- um **conjunto de símbolos não-terminais**, que são variáveis sintáticas e denotam um conjunto de cadeias;
- um **símbolo inicial**, que é um símbolo não-terminal cujo conjunto de cadeias é a linguagem gerada pela gramática
- e um **conjunto de regras de produção**, que especificam como os símbolos terminais e não-terminais podem ser combinados para formar cadeias.

Cada regra de produção é formada por

- Um símbolo não-terminal, chamado **cabeça** ou **lado esquerdo** da regra. A regra define algumas das cadeias denotadas pela cabeça.
- Uma cadeia de zero ou mais símbolos terminais e não-terminais, chamado de **corpo** ou **lado direito** da regra. Os componentes do corpo descrevem uma forma de construir cadeias do conjunto de cadeias do não-terminal no lado esquerdo da regra.
- O símbolo $\rightarrow$ separa os lados esquerdo e direito da regra. Às vezes `::=` é usado no lugar de $\rightarrow$.

Como exemplo considere as regras de produção da gramática 3.1 que define um subconjunto dos comandos e expressões de C. Os símbolos terminais são **`id`**, **`num`**, **`;`**, **`(`**, **`)`**, **`{`**, **`}`**, **`if`**, **`while`** e **`do`**. Nesta

Gramática 3.1: *Subconjunto de comandos e expressões de C.*

Comando	$\rightarrow$ **`id =`** *Exp* **`;`**
Comando	$\rightarrow$ **`if (`** *Expressao* **`)`** *Comando* **`else`** *Comando*
Comando	$\rightarrow$ **`while (`** *Expressao* **`)`** *Comando*
Comando	$\rightarrow$ **`do`** *Comando* **`while (`** *Expressao* **`) ;`**
Comando	$\rightarrow$ **`{`** *Comandos* **`}`**
Comandos	$\rightarrow$ *Comandos Comando*
Comandos	$\rightarrow$
Expressao	$\rightarrow$ **`num`**
Expressao	$\rightarrow$ **`id`**

gramática os símbolos não-terminais são *Comando*, *Comandos* e *Expressão*. O símbolo inicial é *Comando*. Na regra de produção

Comando $\rightarrow$ **`while (`** *Expressao* **`)`** *Comando*

o lado esquerdo é *Comando* e o lado direito é **`while (`** *Expressao* **`)`** *Comando*.

3.2.1 Derivações

Todas as cadeias de uma linguagem podem ser geradas através de derivações a partir do símbolo inicial da gramática da linguagem. As regras de produção são tratadas como regras de reescrita. Fazer uma **derivação** (denotada pelo símbolo *Rightarrow*) em uma cadeia consiste em reescrever a cadeia substituindo um símbolo não-terminal da cadeia pelo lado direito de alguma regra de produção cujo lado equerdo é este símbolo não-terminal.

Por exemplo, considere a gramática 3.1. A seguinte sequência de derivações mostra que a cadeia

"if(id)id=numelseid=id"

pertence à linguagem da gramática.

Comando ⇒ **while (*Expressao*) *Comando***
⇒ **while (*Expressao*) { *Comandos* }**
⇒ **while (*Expressao*) { *Comandos Comando* }**
⇒ **while (id) { *Comandos* *Comando* }**
⇒ **while (id) { *Comandos Comando Comando* }**
⇒ **while (id) { *Comando* *Comando* }**
⇒ **while (id) { id "="*Expressao* ; *Comando* }**
⇒ **while (id) { id "="num ; *Comando* }**

3.3 Expressões aritméticas

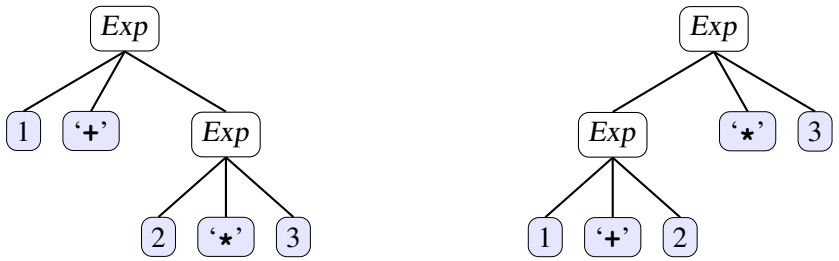
Considere a gramática na tabela 3.1 para expressões aritméticas.

Tabela 3.1: *Gramática de uma linguagem de expressões aritméticas: versão 1.*

$Exp \rightarrow$ **num**
$Exp \rightarrow$ **id**
$Exp \rightarrow$ id **:=** Exp
$Exp \rightarrow Exp$ **+** Exp
$Exp \rightarrow Exp$ **−** Exp
$Exp \rightarrow Exp$ ***** Exp
$Exp \rightarrow Exp$ **/** Exp
$Exp \rightarrow$ **(** Exp **)**

Esta gramática é ambígua, uma vez que é possível construir mais de uma árvore de derivação para algumas sentenças da linguagem, como mostra a figura 3.3.

Figura 3.3: *Árvores de derivação para a cadeia* $1+2*3$.



(a) *Adição é a operação principal*

(b) *Multiplicação é a operação principal*

Por este motivo esta gramática não é adequada para implementação a linguagem de expressões. Vamos escrever uma nova gramática para esta linguagem para remover a ambiguidade. Para tanto vamos definir a prioridade usual dos operadores aritméticos, e designar para a atribuição uma prioridade menor do que todas as demais. O resultado é a gramática da tabela 3.2.

Com esta gramática, a única árvore de derivação possível para a cadeia $1+2*3$ é mostrada na figura 3.4.

Mas esta gramática ainda ainda não é adequada para análise sintática descendente recursiva (ou preditiva), pois existem regras com recursividade à esquerda. No entanto a recursividade à esquerda pode ser facilmente eliminada, levando à gramática da tabela 3.3.

Com esta gramática a árvore de derivação para a cadeia $1+2+3$ é exibida na figura 3.5.

Tabela 3.2: *Gramática de uma linguagem de expressões aritméticas: versão 2.*

Exp → id **:=** *Exp1*
Exp → *Exp1*
Exp1 → *Exp1* **+** *Term*
Exp1 → *Exp1* **-** *Term*
Exp1 → *Term*
Term → *Term* ***** *Fator*
Term → *Term* **/** *Fator*
Term → *Fator*
Fator → **num**
Fator → **id**
Fator → **(** *Exp* **)**

Figura 3.4: *Árvore de derivação para a cadeia* $1+2*3$.

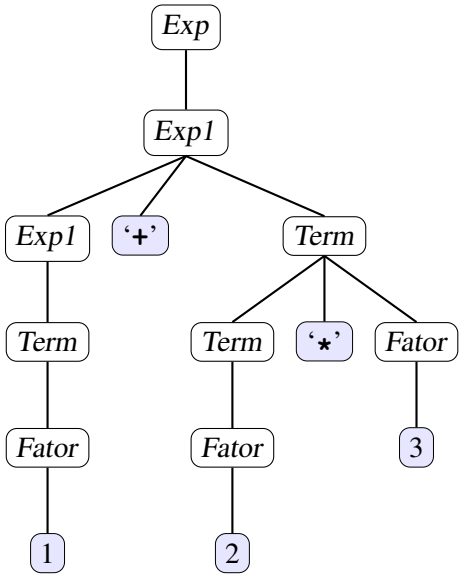
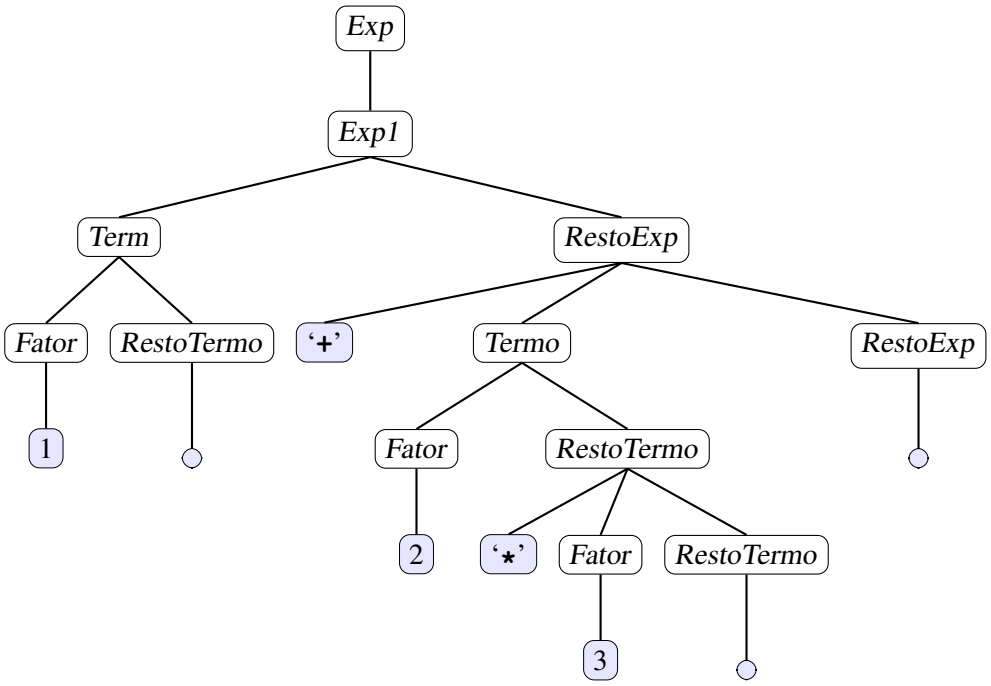


Tabela 3.3: *Gramática de uma linguagem de expressões aritméticas: versão 3.*

Exp → id **:=** *Exp1*
Exp → *Exp1*
Exp1 → *Term RestoExp*
RestoExp → **+** *Term RestoExp*
RestoExp → **-** *Term RestoExp*
RestoExp →
Term → *Fator RestoTermo*
RestoTermo → ***** *Fator RestoTermo*
RestoTermo → **/** *Fator RestoTermo*
RestoTermo →
Fator → **num**
Fator → **id**
Fator → **(** *Exp* **)**

Figura 3.5: *Árvore de derivação para a cadeia* $1+2*3$.



4

Análise sintática de *Tiger*

Exercício 4.1. Utilizando o CUP, implementar um analisador sintático para a linguagem Tiger, definida no apêndice do livro do Appel [1]. Utilize os arquivos disponibilizados em `http://www.iceb.ufop.br/decom/prof/romildo/cic220/praticas/tiger-parser-0.1.rar` para criar uma aplicação em Java para fazer a análise sintática de programas em Tiger. Estão incluídas todas as classes necessárias. Falta apenas completar a gramática livre de contexto para a linguagem Tiger, que será processada pelo CUP.

Referências Bibliográficas

[1] Andrew W Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1998.