# Flex

O flex gera um analisador léxico dividindo o ficheiro de linguagem em texto estruturado e em tokens (símbolos significativos). O flex é, portanto, uma ferramenta para gerar scanners (analisadores léxicos), que reconhecem padrões léxicos num determinado texto, lendo uma entrada (ficheiros de entrada ou input convencional).

Esta descrição é composta por regras, que constituem pares de expressões regulares e código C. As expressões regulares são uma sequência de carateres que formam um padrão de busca, utilizado para correspondência de *strings*, ou pares de *strings*, ou seja, como uma operação de "busca e reposição". O flex após definir as regras léxicas com as respetivas expressões regulares, dividindo um ficheiro de entrada em peças, retorna um símbolo léxico por cada expressão encontrada.

O flex e bison, como referido anteriormente necessitam de duas fontes de código, o ***Source flex***e o ***Source bison***. O código *Source flex* utilizado no projeto é denominado de “**tp.l**” e representa as ações e expressões especificadas pelo utilizador, ou seja, as expressões regulares especificadas dadas depois ao Lex.

Com base num ficheiro fonte (*source code*) escrito de acordo com a sintaxe do flex, o programa flex gerará um analisador léxico descrito na linguagem C. Em caso de existirem erros de codificação, o flex gerará uma listagem de erros. O ficheiro fonte em C terá de ser compilado para a plataforma em utilização utilizando um compilador da linguagem C adequado (neste caso o GCC). O resultado final da compilação será um programa executável capaz de identificar os padrões definidos pelo programador e levar o conjunto de ações previsto. Como entrada para o analisador gerado podem ser fornecidos ficheiros de texto ou alternativamente fornecer os dados diretamente pelo standard de entrada.

O formato utilizado no ***Lex Source***do projeto (“**tp.l**”) apresenta o formato geral deste tipo de códigos, ou seja:

**%{**

**Declarações em C**

**%}**

**Definições**

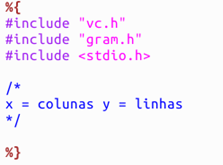
**%%**

**Regras**

**%%**

**Sub-rotinas**

Apesar de muitas vezes as “Definições” ou as “sub-rotinas” estarem omitidas, no caso concreto do projeto, foi necessário definir algumas bibliotecas e incluir alguns ficheiros com estruturas importantes para a compilação e obtenção dos resultados. As declarações em C utilizadas são mostradas na seguinte imagem.



*Figura 1 - As "Definições" declaradas no Source flex.*

O formato da estrutura apresentada apresenta separadores (%%) entre “Definições”, “Regras” e “Sub-rotinas”. Após as “Definições” vem a secção mais importante do ficheiro léxico, as “Regras”. As Regras representam decisões de controlo do utilizador, estão estruturadas como uma tabela, em que a coluna da esquerda contém expressões regulares e a coluna direita contém as respetivas ações, partes do programa a ser executado quando as expressões são reconhecidas.

No código que se segue é possível ver alguns exemplos de regras implementadas no projeto, com o objetivo de reconhecer algumas expressões características da linguagem em Texto Estruturado (ST).

A seguinte imagem apresenta um exemplo de uma expressão regular e respetivo token associado, neste caso a definição de um token correspondente à palavra PONTO.



Figura 2 - Exemplo de uma expressão regular e respetivo "token" associado.

O formato geral do código fonte do lex é:

**Definições**

**%%**

**Regras**

**%%**

**Sub-rotinas**

A entrada do lex, ou seja, o código fonte (***source***) é divido em três secções com "%%" como divisores. O mínimo absoluto que se pode ter num programa em lex é só uma divisão com "%%", (sem definições nem regras), onde traduz para o programa a entrada numa saída inalterada. No formato geral dos programas em lex, as **regras** representam as decisões de controlo do utilizador, correspondendo a duas colunas, onde a coluna da esquerda contém as expressões regulares e a coluna da direita, contém as ações.

As **ações** são fragmentos do programa para serem executados quando as expressões são reconhecidas.

As **expressões regulares** podem conter carateres de texto (que fazem ligar os carateres correspondentes às ***strings***as serem comparadas) ou até carateres de operação (que especificam repetições, escolhas, e outras características). As letras do alfabeto e os dígitos, são sempre considerados carateres de texto.

Os ***tokens*** são representações numéricas de ***strings***que simplificam o processo. Uma expressão regular especifica um conjunto de ***strings***para serem correspondidas em ***tokens*.**

***Caracteres* Significado**

. Qualquer caracter exceto o \n

\n Nova linha

\* Zero ou mais cópias da expressão precedente

+ Um ou mais cópias da expressão precedente

? Expressão opcional

^ Início da linha

$ Fim de linha (*end of line*)

a | b **a** ou **b**

(ab)+ Um ou mais cópias de **ab** (*grouping*)

"a + b" Literal **a+b**

[ ] Classe de caracteres

Tabela 1 - Primitivas de correspondência padrão em expressões regulares.

Quando uma expressão escrita é correspondida, o Lex executa a ação correspondente. Sempre que o utilizador, utilizando o Lex, deseja absorver toda a entrada, sem produzir qualquer saída, então, deve fornecer regras para combinar com tudo. A ação padrão, como já foi referido, consiste em copiar a entrada para saída, cujas strings não são correspondidas. A situação mais normal e comum é o Lex ser utilizado com o Yacc. Neste caso, existe a construção de ações em vez de se copiar a entrada para a saída e, em geral, uma disposição que meramente copia normalmente é omitida.

Uma das coisas mais simples que se pode fazer para se ignorar o input é especificar em código C, a declaração de nulo. A regra frequente que provoca este resultado é "[\t\n ]", que faz com que três carateres de espaço, espaços, tabulações e mudanças de linha sejam ignorados.

Tabela 2 - Exemplos de aplicações padrão em expressões regulares.

***Expressão* Correspondência**

abc\* ab abc abcc abccc ...

abc+ abc abcc abccc ...

[a-z] Qualquer letra de **a-z**

[-az] Uma destas letras: **-, a, b**

[^ab] nada exceto: **a, b**

[a ^b] Um de: **a, ^, b**

[a|b] Um de: **a, |, b**

a | b **a** ou **b**

[ \t \n]+ Espaço em branco

[A-Za-z0-9]+ Um ou mais caracteres alfanuméricos

Para se perceber melhor a utilização prática de expressões regulares e respetivas ações, vai ser mostrado um exemplo simples da aplicação no trabalho prático. Este exemplo é dos mais comuns, porque a abordagem é bastante simples e percebe-se bem o seu funcionamento e a lógica envolvida. O código da imagem ilustra a aplicação do trabalho prático.



Figura 3 - Excerto do ficheiro source do Lex para o trabalho prático

O código mostrado na anterior imagem está dividido em duas partes principais da estrutura formal de um ficheiro flex (separadas pelos carateres %%). A primeira parte, dirigida às definições, especifica um conjunto de expressões regulares que, com o acoplamento ou conjugação de umas com as outras, vai definindo representações de vários tipos de dados.

Na parte das regras, vão ser associadas as verdadeiras ações do flex, ou seja, para cada tipo de entrada reconhecido, vai ser executado uma instrução em código C que desempenhe a ação premeditada pelo utilizador. Neste exemplo, e porque vai haver uma ligação flex e bison, todas as expressões reconhecidas e especificadas nas regras, são retornadas com o respetivo ***token***associado.

O flex pode ser utilizado sozinho para transformações simples em nível léxico, tal como o exercício prático nº 2 que consistia em fazer alterações no texto onde eram apresentadas abreviaturas e era proposto a alteração das mesmas para a palavra equivalente da abreviatura, mas pode também ser utilizado com um gerador (***scanner***), para executar a fase de análises léxicas.

Quando o analisador léxico encontra identificadores nos fluxos de entrada, coloca-os numa Tabela de símbolos. A tabela de símbolos pode conter também, outra informação tal como o tipo de dados (inteiro ou double) e a localização de memória de cada variável. Todas as referências subjacentes aos identificadores, referem-se ao índice da tabela de símbolos apropriado.

Quando um analisador léxico retorna um fluxo de *tokens* cada *token* está dividido em duas partes, o número do *token* e o valor do *token*. No fundo, um *token* é um pequeno número. Todos estes números são arbitrários, exceto o número zero que está reservado para o "*end\_of\_line*".

Quando o *bison* (versão moderna do Yacc) cria um *parser* (analisador sintático), atribui os números dos *tokens* automaticamente começando no 258 (evitando colisão com outros carateres que acabam no 257), criando um "gram.h" com as definições destes números. Um valor de *token* identifica a que grupo de *tokens* similares esse *token* pertence.

A variável "yylval" armazena o valor do *token*, ou seja, um inteiro. Mais tarde veremos que o valor que o yylval poderá ter outro tipo de valor com a ajuda de um *union*.

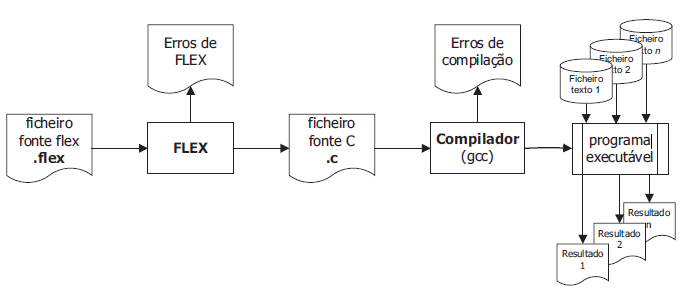


Figura - Ciclo de vida de um programa FLEX

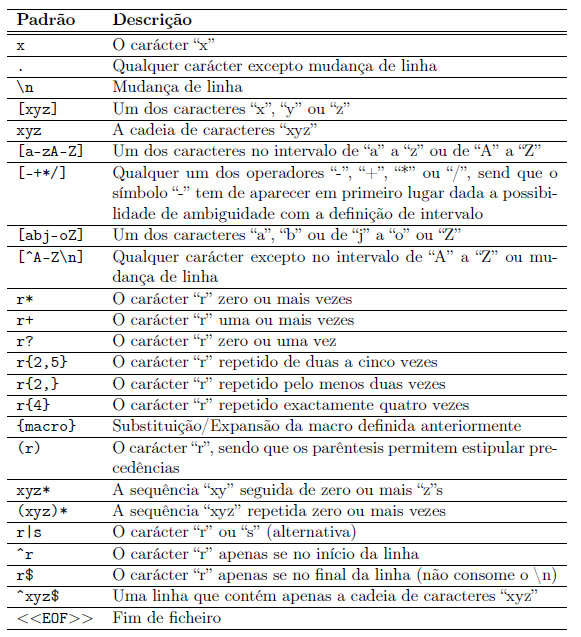


Figura - Primitivas de correspondência padrão em expressões regulares.

# Bison

O Bison é um descendente de yacc (yet another compiler compiler), um gerador de um analisador sintático (parser), criado entre 1975 e 1978 por Stephen C. Johnson na Bell Labs. Como o próprio nome afirma, a abreviação de "mais um compilador de compilador", indica que muitas pessoas estavam a criar ou escrever analisadores sintáticos na altura.

Richard Stallman da fundação FSF (Free Software Foundation), adaptou o trabalho de Corbett para utilizá-lo no projeto GNU, onde foi crescendo para aumentar um vasto número de novas funcionalidades, até evoluir na versão presente do Bison.

O Bison é agora mantido como um projeto FSF e é distribuído como uma licença pública GNU. Em 1975, Mike Lesk e Eric Schmidt escreveram o lex, um gerador de análise léxica. Viram-no como uma ferramenta autónoma e como um complemento e "companheiro" do yacc desenvolvido por Johnson.

O lex tornou-se também bastante popular, apesar de inicialmente ser relativamente lento e conter alguns bugs.

GNU é um sistema operativo de computador em Unix desenvolvido pelo GNU Project 3Bugs, em informática, são conhecidos como erros de um programa de computador.

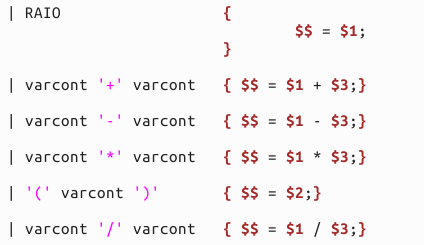
Schmidt, no entanto, passou a ter uma carreira bastante bem-sucedida na indústria de computadores, onde chegou a ser CEO do Google até abril de 2014. Em 1987, Vern Paxson, do laboratório de Lawrence Berkeley, fez uma versão escrita do lex em "ratfor" (uma versão Fortran prolongada) e traduziu-a para a linguagem C, chamando-a de Flex (Fast Lexical Analyzer Generator). Desde que ficou mais rápido e confiável que o lex AT&T, ultrapassou completamente o lex original, sendo agora um projeto SourceForge, sob a licença Berkeley.

O *Bison* vai fazer a segunda parte do trabalho, sendo responsável pela análise sintática e semântica, dentro das etapas definidas por um compilador. O *Bison* ao receber os *tokens* vindos do *Flex*, agrupa-os logicamente, e define toda a gramática predefinida pelo utilizador, através de um ficheiro yacc com as respetivas regras. As regras de produção criadas, vão levar à criação de ações, cuja implementação, normalmente é definida por funções programadas em linguagem C.

O bison, através desses símbolos recebidos, forma as regras de produção e a respetiva estrutura hierárquica do programa.

Em processos de tradução de programas *source* para programas alvo, os compiladores devem construir pelo menos uma representação intermediária, como árvores de derivação e respetivamente árvores sintáticas, após a análise léxica (através do *lexical analyzer*) e análise sintática (através do *parser*). A criação de código numa linguagem alvo é realizada através da entrada duma representação intermediária da linguagem do programa de origem (*source*) no *code generator*, saindo a linguagem alvo desejada (*target*).

São as ações implementadas, que especificam o significado e a semântica das regras gramaticais definidas pelo ficheiro de entrada bison. A imagem seguinte apresenta um pequeno conjunto de aplicação de regras de produção.



As especificações bison têm a mesma estrutura tripartida que as especificações flex. A primeira secção é a secção das definições, que lida com o controlo de informação para o *parser* e, geralmente, configura o ambiente de execução em que o analisador (*parser*) irá operar, ou seja, consiste nas declarações de variáveis globais, incluir as bibliotecas e os ficheiros anexos à estrutura do programa (ficheiros .h, ficheiros .c, etc) e ainda depois de fechar as chavetas declaramos os tokens, union, types, e/ou outras keyword do bison. A segunda secção contém a descrição das regras gramaticais para a especificação da linguagem. A terceira secção contém as sub-rotinas definidas em código C.

O formalismo da gramática está dividido em três tipos de notações:

* + - * Símbolos não-terminais, são sempre escritos em minúsculas;

**– (**e.g. varcont, retfill, ciclo, linha, retângulo)

* + - * Símbolos terminais (ou tokens), são escritos em maiúsculas ou por carateres únicos;

**–** (e.g. RAIO, VAR, PONTO,’;’,’ (’)

* + - * Regras gramaticais, produção de regras;

**–** (e.g. varcont: varcont ’+’ varcont | varcont ’ \* ’ varcont;)

Os símbolos terminais (ou ***tokens***) são uma classe de elementos equivalentes, do ponto de vista sintático. São representados por códigos numéricos associados aos seus identificadores.

O ***scanner***léxico "yylex()" deve retornar o código correspondente à classe do elemento correspondido no texto de entrada;

Os símbolos terminais são declarados na secção de Definições Bison;

* + %token PONTO

Os *tokens* literais são utilizados para corresponder constantes de carateres em C (e.g.’+’,’\*’,’-’), não sendo preciso declara-los explicitamente, a menos que seja necessário especificar os tipos de dados associados, a sua precedência ou a sua propriedade associativa (*left/right*) para simplificar a gramática.

O operador de precedência depende da ordem das declarações:

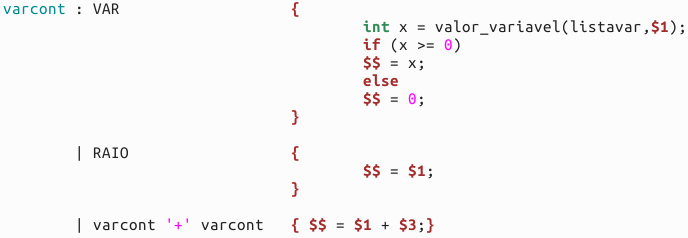
%left ’+’ ’-’ (menor precedência)

%left ’\*’ ’/’ (maior precedência)

As regras gramaticais seguem a seguinte estrutura:



O "varcont" é o símbolo não-terminal para qual o lado direito da regra de produção é reduzida. O lado direito da regra de produção é a sequência de componentes que tanto podem consistir em símbolos terminais, como em não-terminais ou ações (código C entre {...}), por exemplo:



O símbolo ’+’ é um símbolo terminal e a ação correspondente a esta regra é "$$=$2+$3;". Cada expressão termina com uma nova linha. Quando a nova linha é detetada é impresso o valor da expressão. O lado direito da produção é substituído na pilha (*stack*) do *parser* pelo lado esquerdo da mesma produção.

Neste caso, a regra “varcont ‘+’ varcont" é retirada (*popping*) da pilha e a regra "varcont" é colocada (*pushing*) na pilha.

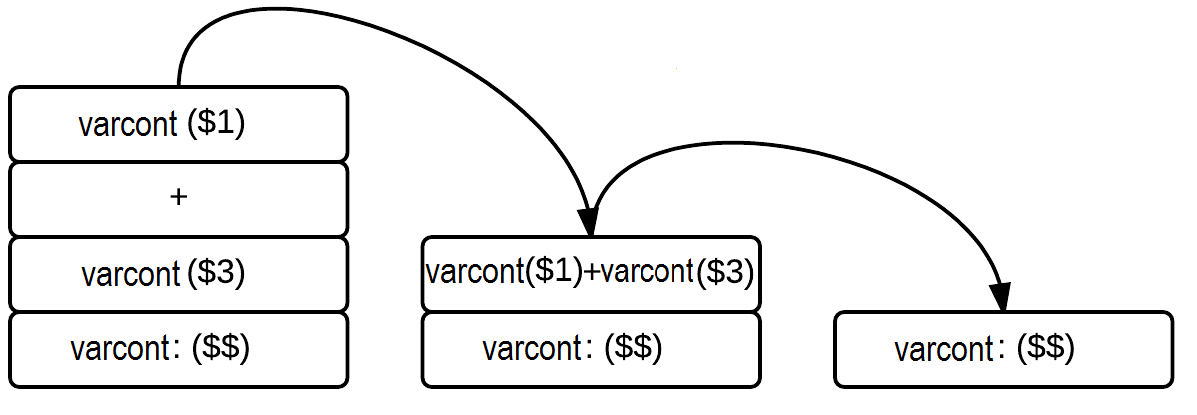


Figura - Exemplo de um processo de redução de pilha através de um "LR parser".

Este raciocínio faz com que seja reduzida a pilha, destruindo três termos e colocando um. As posições na pilha de valores na parte do código C {$$=$1+$3;}. O termo "$$" corresponde ao topo da pilha depois de a redução ter sido feita. Sendo assim, a ação descrita, soma o valor associado a duas expressões, substitui três termos da pilha por uma soma simples, tornando o *parser* e a pilha de valores (*stack*) sincronizados.

Se diferentes tipos de dados são para ser utilizados por diferentes símbolos, a lista completa dos tipos de dados deve estar especificada nas declarações bison. Um dos tipos de dados declarados no código está associado ao símbolo terminal (%token) e não-terminal (%type).

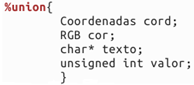


Figura - Exemplo da utilização de um %union para a definição de diferentes tipos de dados

Os diferentes tipos de dados utilizados para diferentes símbolos podem ser acoplados numa lista (union), ocupando cada um deles o mesmo slot de memória, não podendo ser utilizados em simultâneo. O símbolo terminal, como o caso do “%token <valor> RAIO", corresponde a um valor inteiro como tipo de dado, e o símbolo não terminal, como o caso do “%type <cor> color”, corresponde a uma declaração de uma estrutura de dados que dará suporte ao formato RGB.



Figura - Excerto do ficheiro de Bison com respetivas regras gramaticais e de produção

Como se pode observar, o ficheiro bison correspondente às regras gramaticais do trabalho prático, onde o qual está dividido pelas três zonas principais da forma geral, que não é possível observar na anterior imagem. A primeira zona das definições contém as declarações de código C necessário (código entre % {. . . }%), como a inclusão de bibliotecas, a definição do tipo de dados do "yylval", e as declarações dos *tokens* e respetivas prioridades e propriedades associativas.

Na secção das regras de produção estão definidas as regras que vão originar a árvore de derivação e a árvore sintática.

Como já tinha sido referido, o bison utiliza as regras gramaticais, permitindo-lhe analisar os *tokens* vindos do flex e criar a árvore de sintaxe.

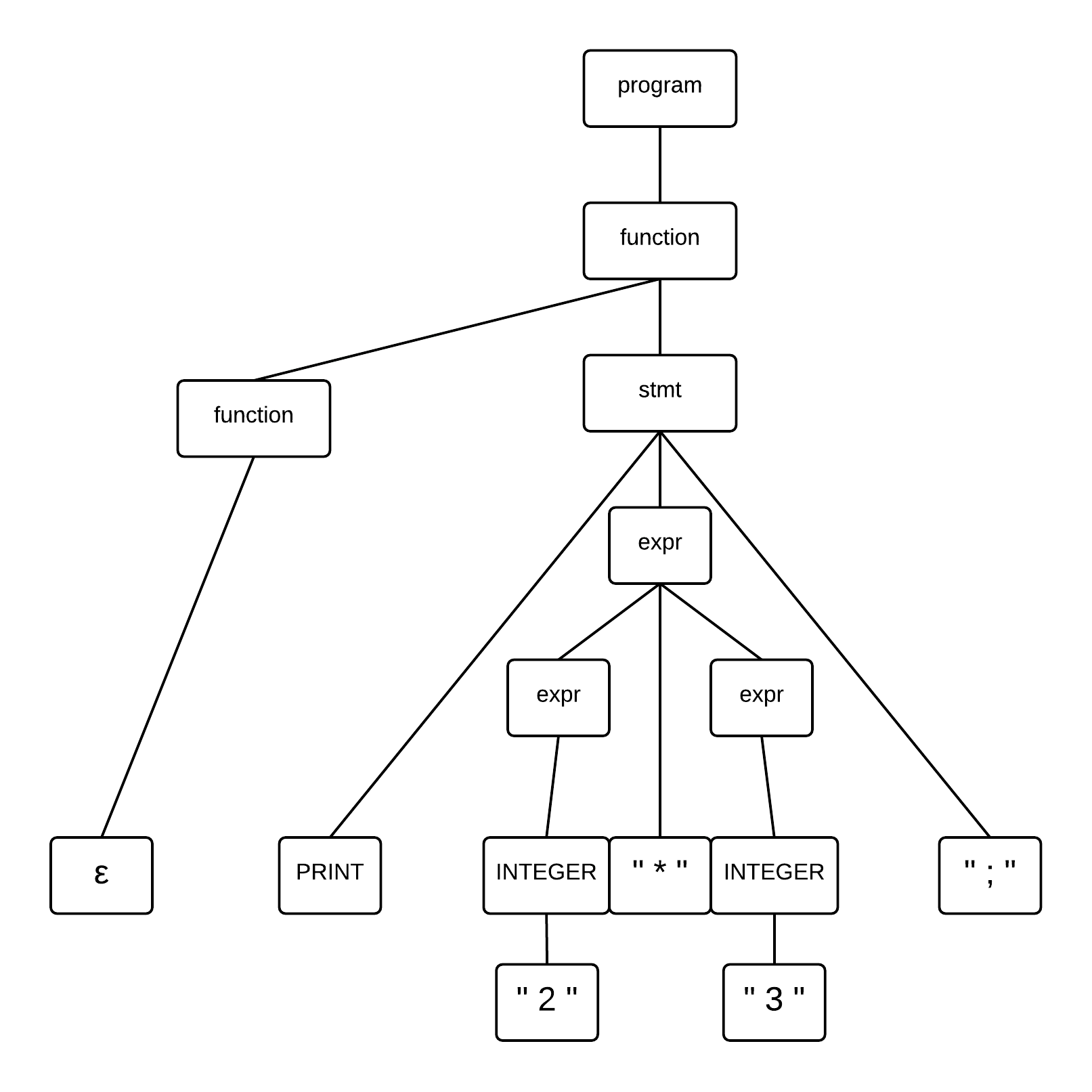


Figura - Exemplo de uma árvore de derivação para uma calculadora

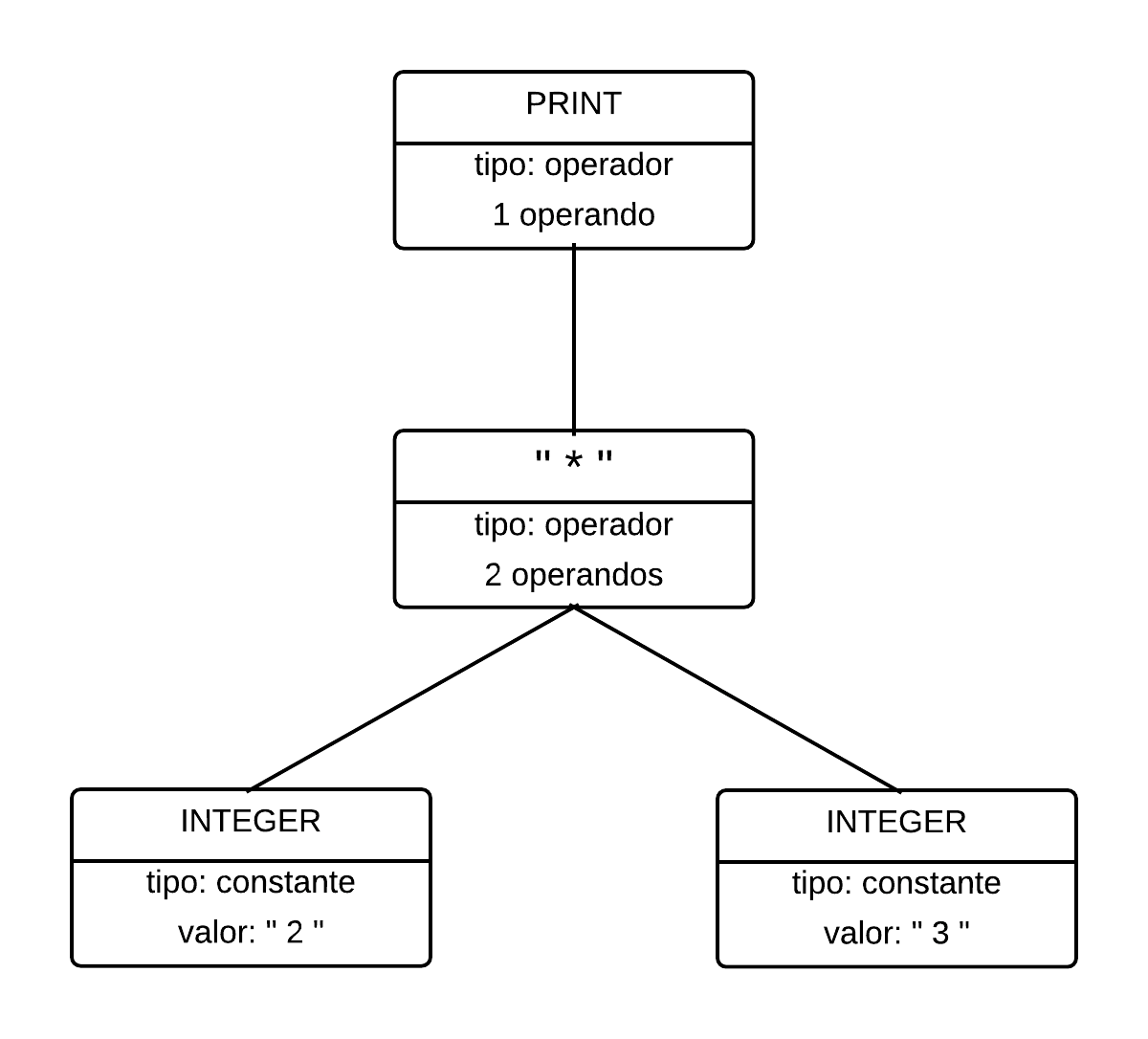


Figura - Exemplo de uma árvore sintática.

# Compilador e interpretador

Os diversos serviços e pessoas ligadas, diretamente ou indiretamente à área da informática precisa e depende das linguagens de programação, porque o *software* que corre nos sistemas dos computadores, está escrito em alguma linguagem de programação. O processo que permite um programa correr sem problemas, não é assim tão simples e direto, sendo necessário, ser traduzido numa forma e estrutura que possa ser executado por um computador, denominado de código máquina.

O sistema de *software* que permite fazer uma translação e tradução de linguagens de programação é o **compilador**. O compilador pode ler um programa numa linguagem (*source program*) e traduzi-la num programa equivalente noutra linguagem, tendo como importante característica, o facto de conseguir reportar erros do programa fonte (*source*) que são detetados durante o processo de tradução.

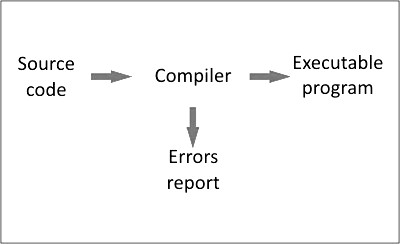


Figura - Um compilador

Um *interpreter* não é muito diferente de um compilador, convertendo também uma linguagem de alto nível. Cada vez que o *interpreter* recebe o código de linguagem de alto nível a ser executado, converte-o num outro código intermediário antes do código da máquina. O código é executado separadamente numa sequência, encontrando, se for o caso, erros associados na sua interpretação. Sempre que encontra um erro, para a interpretação e não traduz o próximo conjunto de códigos. As principais diferenças entre um compilador (*compiler*) e um interpretador (*interpreter*) estão resumidas na próxima tabela:

**Interpretador (*Interpreter*)**

O *interpreter* agarra num conjunto de instruções (*statement*), tradu-lo, executa-o e por fim agarra outro conjunto.

O *interpreter* vai parar a tradução depois de encontrar o primeiro erro.

O *interpreter* leva muito menos tempo a analisar o processo do código de linguagem de alto nível.

**Compilador (*Compiler*)**

O *compiler* traduz o programa inteiro num outro e então, executa-o.

O compilador gera um relatório de erro

depois da tradução do programa inteiro

Para o mesmo processo, o compilador leva muito mais tempo a realizá-lo.

Tabela 3 - Principais diferenças entre compilador e interpretadores

Colocando de lado o tempo de processamento e análise, o tempo geral de toda a execução do código, é mais rápida no compilador do que no interpretador.

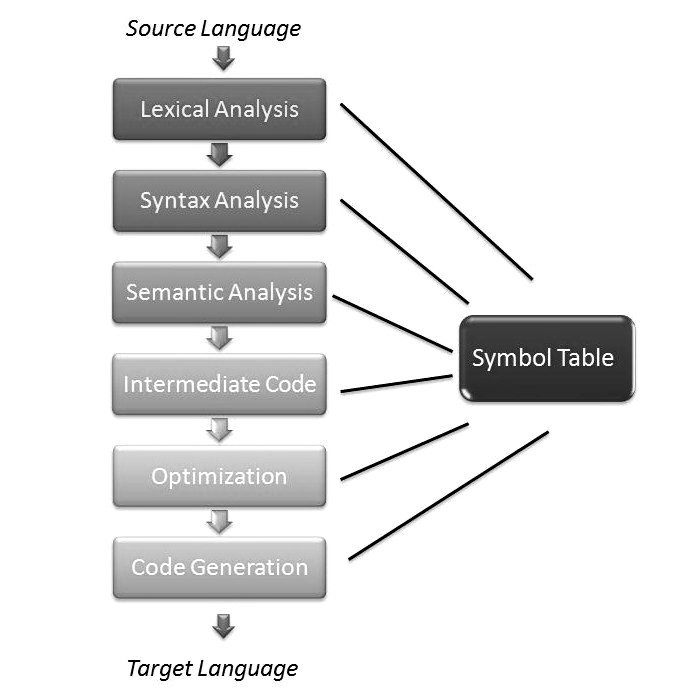


Figura - Fases de um compilador

Como se pode ver pela imagem a primeira fase do compilador corresponde à **análise léxica** (*lexical analysis or scanning*), em que o analisador lê uma sequência de carateres e converte-os numa sequência de ***tokens.*** A análise léxica funciona como um verificador de um alfabeto, inventado pelo utilizador ou não, em que se verifica se determinado caráter existe ou não nesse alfabeto.

A segunda fase do compilador é a **análise sintática** (*syntax analysis or parsing*). O *parser* utiliza os *tokens* produzidos pelo analisador léxico para criar tipos de representações intermediárias de árvores (árvores de derivação), que definem a estrutura gramatical das sequências de *tokens*.

Estas árvores de derivação (*parse trees*) são a representação gráfica de uma derivação, que mostra a estrutura hierárquica do programa (*source program*).

As seguintes imagens ilustram o processo e as etapas desde o programa *source* até à estrutura sintática.

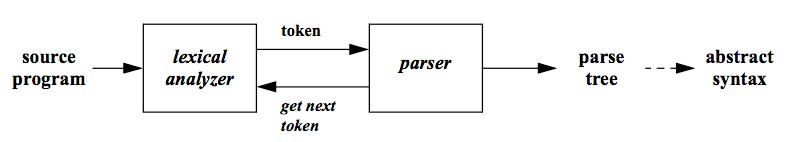


Figura - Processo de construção da estrutura sintática.

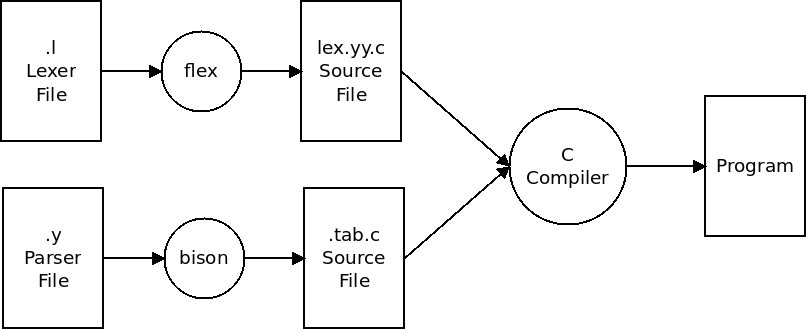


Figura - Processo geral de compilação Flex/Bison.

# Makefile

### História

Em 1977, [Stuart Feldman](http://en.wikipedia.org/wiki/Stuart_Feldman) criou o primeiro sistema de automação para a compilação de programas, o [make](http://en.wikipedia.org/wiki/Make_%28software%29). A função do make é, construir todas as dependências descritas num ficheiro chamado Makefile. Makefiles seguem um padrão razoavelmente simples de construção. Embora o seja relativamente acessível a criação do Makefile, muitos IDE’s trazem consigo o seu próprio make, possuem as suas próprias regras e mecanismos, baseado ou não no make do UNIX.

### O que é a makefile?

O Make é um programa de computador que tem o intuito de automatizar a compilação de programas que usam diversos ficheiros. As instruções que o Make executa estão todas dentro de um ficheiro chamado Makefile (ou makefile).

### Porque foi usado?

Pensamos que seja uma ferramenta indispensável para qualquer programa que se faça, pois, a compilação é das partes mais chatas do desenvolvimento e de testes de um programa, então a automatização destas tarefas de rotina irá facilitar a compilação tal como a vantagem da poupança de tempo em comparação com a execução de comando a comando.

O texto contido em um Makefile é usado para a compilação, ligação, montagem de arquivos de projeto entre outras tarefas como limpeza de arquivos temporários, execução de comandos.

# Curiosidades

Processamento de linguagens natural nasceu com o computador:

2ª Guerra: militares americanos tinham interesse de traduzir automaticamente conversações gravadas dos russos;

A comunidade científica precisava de traduções de trabalhos estrangeiros, então a cada dia fazia-se novas descobertas científicas.  
  
O Lex e Yacc, ou versões melhoradas e mais rápidas Flex e Bison respetivamente, são os softwares correspondentes que permitem gerar estes vários tipos de analisadores em linguagens de programação.