Opções do SLK

Nuno Ernesto Salgado Oliveira
17 de Novembro de 2007

1 Introdução

O slk (**S**trong **LL**(**K**))é um gerador de parser que contém vários comandos de opções, cuja maioria se traduz em *output* importante para análise da gramática escrita.

o slk Pode ser invocado a partir da linha de comandos[1] sem qualquer tipo de opção especificada:

$$slk \ < nome\_ficheiro > \tag{1}$$

O *output* base que surge deste comando é uma simples lista que informa o utilizador sobre:

1. **Número de não–terminais** basicamente são contados os símbolos que ficam do lado esquerdo do símbolo reservado ':'.
2. **Número de terminais** neste caso são contados todos os símbolos terminais da gramática, ou seja, o número de símbolos que aparecem do lado direito so símbolo ':' e não aparecem do lado esquerdo.

   Contudo são ainda acrescentados símbolos terminais usados para construção da gramática, como é exemplo o símbolo "END_OF_SLK_INPUT".
3. **Número de produções** são contadas as produções.
4. **Tamanho da tabela** é o tamanho da tabela de parsing para parsers *LL*. Em suma é a multiplicação do número de símbolos terminais pelo o número de símbolos não terminais.
5. **Total de conflitos base**
6. **Total de conflitos**
7. **Tamanho da tabela de conflitos**
8. **Resultado da compactação da tabela de parsing**
9. **Resultado da compactação da tabela de conflitos**

Por omissão, com apenas o comando especificado em cima, são criados 7 ficheiros que descrevem o parser em linguagem `C`.

[1] Para um uso mais rápido desta ferramenta, é interessante torná-la acessível a partir de qualquer parte do sistema. Um pequeno tutorial para se conseguir esse objectivo segue na secção **??**

2 As Opções

Como já foi dito, enumeras opções podem ser tomadas para obter diferente *output* sobre a gramática que alimenta o slk. A lista completa dessas opções pode ser vista quando invocado o programa slk sem qualquer argumento.

As opções serão de imediato explicadas.

- **-a**
  Activar esta opção faz acrescentar à definição dos tokens o sufixo `_Slk`. Se apenas se usar este comando, os ficheiros alterados são `SlkKeywords.txt` e `SlkParse.h`. Se se usar esta opção com o comando *-n=XXX*, então o sufixo para os tokens será `_XXX`.

- **-c**
  Esta opção faz com que o slk deixe de compactar as tabelas quer de parsing, quer de conflitos. Acabando por ocupar mais memória. As diferenças entre activar a acção ou não pode ser vista nas últimas linhas do *output* gerado pelo comando 1, onde mostra o resultado de compactar as tabelas de parsing e de conflitos. Os primeiros valores são das tabelas não compactadas, os últimos referem-se ao resultado da compactação.

  Como na maior parte das vezes, as entradas destas tabelas são situações de erro, então não vale a pena ocupar memória com essa informação desnecessária. Assim a compactação consiste em guardar as situações em que não há erro.

- **-cs**
  Esta opção instrui o slk para criar como *output* ficheiros que descrevem o parser gerado em `C#`, deixando de parte aqueles que eram gerados por omissão em C.

- **-C++**
  À imagem da opção anterior, faz o slk gerar ficheiros que descrevem o parser em `C++`.

- **-Da**
  Esta opção é muito generalista. A sua função é mostrar no ecrã da linha de comandos todo o *output* que é possível mostrar no ecrã: Métricas de tamanho da gramática, Lookahead para os conflitos resolvidos, gramática reescrita em texto e em inteiros, tabela de parsing, tabela de conflitos, First dos não–terminais, Follow dos não–terminais, não–terminais que derivam em ϵ, etc.

- **-Dc**
  Esta opção faz com que o slk mostre no ecrã o trace dos conflitos, ou seja, quando há conflitos na gramática, esta opção faz mostrar no ecrã quais os não–terminais onde se dá conflito, em que símbolo terminal, e qual o número do conflito, e qual o nível de lookahead a que o conflito é resolvido.

- **-Dd[=nome_símbolo]**
  Esta opção mostra uma lista de todos os não–terminais que derivam à esquerda o símbolo nome_símbolo.

- **-De**
  Instrui o slk para criar uma lista de todos os não terminais que derivam na palavra vazia, isto é, em ϵ.

- **-Df**
  Faz mostrar no ecrã o conjunto *first* e *follow* de cada um dos não–terminais: Numa primeira listagem mostra o first dos não–terminais; este conjunto não é de símbolos terminais, mas sim de símbolos não–terminais. A segunda listagem refere-se ao *first* de todos os não–terminais, desta vez a lista é de símbolos terminais. A última listagem refere-se ao *follow* dos símbolos não terminais.

- **-Dl**
  Esta opção faz com que o slk apresente no ecrã as strings que permitiram resolver os conflitos LL para cada não–terminal. O tamanho da string irá depender do tamanho do valor do k, ou seja do número de símbolos que o parser terá que ler para resolver o conflito.

- **-Dp**
  Instrui o slk para mostrar no ecrã as produções da gramática. Repare-se que a gramática que é dada como *input* pode ser algo diferente da gramática mostrada aqui. Isto acontece essencialmente quando a gramática de entrada é especificada usando a notação EBNF suportada pelo slk. As transformações das gramáticas poderão ser vistas na secção 3, mais à frente.

- **-Ds** Mostra no ecrã uma tabela com todos os símbolos que fazem parte da gramática, e ainda alguns que são usados pelo slk para processamento.

  Na primeira coluna aparecem os valores de hash criados para identificar univocamente cada um dos tokens.

  Na segunda coluna é mostrado o tipo dos símbolos. Os valores possíveis de encontrar aqui são o 1 e o 2, que representam o tipo "Não–Terminal" e "'Terminal" respectivamente.

  Na terceira coluna é mostrado o valor interno do token que o slk atribui. Repare-se que para símbolos reservados, como é o caso do [, por exemplo, o slk utiliza um valor interno negativo.

  Na quarta coluna vêm os nomes dos símbolos ou a sua representação.

- **-Dt** Gera e mostra as tabelas, quer de parsing, quer de conflitos. Vejamos o seguinte exemplo de uma gramática:

```
 1:  PIs --> RESUMO Lst DETALHE Projs .
 2:  Lst --> InvPs ;_InvPs_*
 3:  InvPs --> SglInv LstIds
 4:  SglInv --> id
 5:  LstIds --> SglProj ,_ListIds_*
 6:  Projs --> Proj Proj_* .
 7:  Proj --> SglProj Desc FINANC Entidade Valor INIC Ano FIM Ano
 8:  SglProj --> id
 9:  Desc --> str
10:  Entidade --> FCT
11:  Entidade --> GRICES
12:  Entidade --> ADI
13:  Ano --> num num
```

```
 14:  Ano --> num
 15:  Valor --> num_opt
 16:  ;_InvPs_* --> ; InvPs ;_InvPs_*
 17:  ;_InvPs_* -->
 18:  ,_ListIds_* --> , ListIds ,_ListIds_*
 19:  ,_ListIds_* -->
 20:  Proj_* --> Proj Proj_*
 21:  Proj_* -->
 22:  num_opt --> num
 23:  num_opt -->
```

Vamos obter as seguintes tabelas:

```
                        PARSE TABLE

PIs:   RESUMO 1
Lst:   id 2
InvPs:   id 3
SglInv:   id 4
LstIds:   id 5
Projs:   id 6
Proj:   id 7
SglProj:   id 8
Desc:   str 9
Entidade:   FCT 10   GRICES 11   ADI 12
Ano:   num -1
Valor:   INIC 15   num 15
;_InvPs_*:   DETALHE 17   ; 16
,_ListIds_*:   DETALHE 19   ; 19   , 18
Proj_*:   . 21   id 20
num_opt:   INIC 23   num 22



                        CONFLICT TABLE

 1:    . 14   id 14   FIM 14   num 13
```

Ambas as tabelas são lidas da mesma forma.

Por exemplo, a primeira linha da tabela de parsing pode ser lida da seguinte forma: "*Estamos no símbolo não–terminal PIs, e estamos a ler o símbolo terminal RESUMO, então usamos a produção número 1 da gramática*"

O conflito na gramática surge nas produções 13 e 14, pois é impossível decidir, sem olhar ao que vem a seguir no texto de entrada, qual das duas produções usar. É por isto que surge o valor -1 na tabela de parsing, quando estamos no não–terminal "Ano" e lemos um terminal "num". Ele indica que temos então que recorrer à tabela de conflitos, à entrada 1, para poder resolver o conflito.

Assim a entrada na tabela de conflitos pode ser lida da seguinte maneira: "*Estamos perante o conflito 1, caso o símbolo que vem a seguir (determinado pelo lookahead) seja um de '.', 'id' ou 'FIM' seguimos pela produção 14, mas se o símbolo é 'num', então seguimos pela produção 13*".

- **-En** Dependendo do valor de n, esta opção faz a transformação de gramáticas IEEE e ISO em gramáticas aceites pela sintaxe do slk. Caso a opção seja -E1, então estamos a querer converter uma gramática IEEE em slk, caso a opção seja -E2 então estamos a querer converter gramáticas ISO em slk.

  Esta conversão deve ser feita em 2 passos:

  ```
  slk ieee_grammar -E1 -f=out_grammar

  slk out_grammar -f=slk_grammar
  ```

  O exemplo de utilização dado acima refere–se a n=1, para n=2 o procedimento é exactamente igual.

- **-e** Esta opção *desliga* o apoio à extensão à notação BNF. Assim, podemos usar os caracteres reservados para a notação EBNF, como sejam [,], {, },}+ sem que a semântica associada a estes símbolos seja aplicada à transformação da gramática como explicado na secção 3. Por exemplo, passa a não ser preciso usar '\[' para se ter '[' como um símbolo da nossa gramática.

- **-f=nome_ficheiro** Esta opção faz o slk reescrever a gramática e criar um ficheiro (nome_ficheiro) como *output*.

  Mesmo que não haja qualquer tipo de transformação a fazer à gramática, o novo ficheiro é criado. Os comentários são sempre limpos.

  Se houver alguma transformação a fazer à gramática, por exemplo, utilizando a opção '-e', se a gramática inicial usa notação EBNF, então a nova gramática (novo ficheiro) deixará de usar essa notação, possuindo o caracter '\' para escapar os caracteres reservados à notação EBNF.

  É necessário ter em atenção quando se utiliza as terminações _opt, _*, ou _+ nos não terminais, pois com '-e' activada, ele vai tentar criar uma nova produção para esses não–terminais, o que pode estar errado.

- **-Ga** Esta opção faz com que o slk não gere qualquer tipo de ficheiro de código como *output*.

- **-Gn** Faz com que não se gerem as strings de não–terminais que são usadas nas mensagens.

- **-Gp** Faz com que não se gerem strings de produções que são usadas nas mensagens.

- **-Gs** Faz com que não se gerem strings de acções que são usadas nas mensagens.

- **-Gt** Faz com que não se gerem strings de terminais que são usadas nas mensagens.

- **-Gu** Faz com que não se gerem nomes únicos quando há não–terminais EBNF duplicados. Por exemplo dando a seguinte gramática como input ao slk:

$$\begin{array}{lcl} A & \rightarrow & \alpha \ \{ \ \psi \ \} \\ B & \rightarrow & \psi \ \{ \ \psi \ \} \end{array}$$

  Ele produziria alterações na gramática como se pode ver a seguir[2] (sem usar a opção em análise):

$$\begin{array}{lcl} A & \rightarrow & \alpha \ \psi\_* \\ B & \rightarrow & \psi \ \psi\_ * 2 \\ \psi\_* & \rightarrow & \psi \ \psi\_* \\ & | & \\ \psi\_ * 2 & \rightarrow & \psi \ \psi\_ * 2 \\ & | & \end{array}$$

  Contudo não exactamente isso que se quer, visto que os não–terminais serão iguais em termos do seu lado direito. Portanto a opção '-Gu' geraria a gramática da seguinte maneira:

$$\begin{array}{lcl} A & \rightarrow & \alpha \ \psi\_* \\ B & \rightarrow & \psi \ \psi\_* \\ \psi\_* & \rightarrow & \psi \ \psi\_* \\ & | & \end{array}$$

- **-i** Instrui o slk para ignorar todos *warnings* que não são fatais.

- **-j** Faz com que o slk gere ficheiros em java para a produção do parser, e deixe de gerar os ficheiros que gerava por omissão em C.

- **-k=valor do k em LL(k)** É o número de símbolos que queremos ler para a frente, no caso de termos uma gramática com conflitos. O parâmetro *valor* é um número inteiro que vai desde 1 até ∞. Apesar de aceitar o 0 como um valor válido, ele assume-o que sendo 1.

  Esta opção informa ainda qual é o grau da força LL que a gramática possui. Se colocarmos -k=2, mas a gramática for LL(1), então o slk informará que a gramática é strong LL(1). Portanto o valor de k é apenas um valor máximo para testar a complexidade da resolução de conflitos.

- **-l=nome_não–terminal** Se temos uma gramática como a seguinte:

$$\begin{array}{lcl} A & \rightarrow & \alpha \ B \mid C \ \beta \\ B & \rightarrow & \psi \\ C & \rightarrow & \alpha \end{array}$$

  então iremos ter conflitos, porque o mesmo lado esquerdo, embora de forma indirecta, é igual para cada uma das opções do não terminal A. Uma solução para resolver este conflito é a seguinte:

[2] Uma melhor explicação sobre estas transformações podem ser vistas na seccção 3.

$$\begin{array}{lcl} A & \rightarrow & \alpha A_1 \\ A_1 & \rightarrow & B \mid \beta \\ B & \rightarrow & \psi \\ C & \rightarrow & \alpha \end{array}$$

É exactamente isto que a opção -l faz, ou seja, faz o *left-factoring* indirecto do não–terminal *nome_não–terminal*. Caso se escolha um não terminal que não esteja nestas condições, o slk não irá fazer nada.

- **-Ld[=nome_não–terminal]** Se temos um não terminal como o do seguinte exemplo:

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ \ldots \ \mid \alpha\beta_n$$

então iremos ter conflitos, porque o mesmo lado esquerdo é igual para cada uma das opções para esse não–terminal. Uma solução para resolver este conflito é *pôr em evidência* aquilo que é igual, e aquilo que é diferente delegar noutro não–terminal, como mostra a seguir:

$$\begin{array}{lcl} A & \rightarrow & \alpha \ A_1 \\ A_1 & \rightarrow & \beta_1 \mid \beta_2 \mid \ \ldots \ \mid \beta_n \end{array}$$

Ora é exactamente isto que a opção -Ld faz, ou seja, faz o *left-factoring* directo do não–terminal *nome_não–terminal*. Caso se escolha um não terminal que não esteja nestas condições, o slk não irá fazer nada.

Note-se que para ver o efeito desta e da opção anterior, é necessário utilizar o comando '-f' para gerar um ficheiro de *output*.

- **-Li[=nome_não–terminal]** Esta opção faz exactamente aquilo que a opção '-l' faz, mas de uma forma mais automatizada, pois pode ser feito de imediato a toda a gramática o *left-factoring* indirecto.

- **-m** Esta opção faz que o slk reutilize estruturas internas para consumir menos memória.

- **-n=nome_parser** Permite acrescentar ao parser, isto é, aos tokens e às funções do parser um sufixo que identifica o nome do mesmo, como já foi referido aquando da explicação da opção '-a'.

  A utilização desta opção permite a possibilidade usarmos vários parsers num só programa. Podendo eles terem o mesmo nome base para os tokens, sem que haja qualquer tipo de conflito.

- **-ns=namespace ou package** Gera os ficheiros C++ ou C# com um namespace, ou os ficheiros Java com uma package associada.

- **-O[=n]** Faz com que slk apenas analise um conflito de cada vez. O valor de n indica qual o conflito a analisar.

- **-o[=n]** Faz com que o slk analise apenas o n–ésimo conflito dentro de um conflito, visto que por cada conflito podem ser gerados vários conflitos por cada nível de lookahead.

- **-q** Instrui o slk para não mostrar muitas mensagens, ou seja, trabalhar num modo silencioso.

- **-Rd** Remove produções que estejam duplicadas na gramática. Por exemplo, numa gramática como a seguinte:

$$\begin{array}{lrcl} p1: & A & \rightarrow & \alpha\ B \\ p2: & B & \rightarrow & \beta \\ p3: & B & \rightarrow & \beta \end{array}$$

  as produções p2 e p3 estão duplicadas, logo a opção '-Rd' iria transformar a gramática anterior na que se mostra a seguir:

$$\begin{array}{lrcl} p1: & A & \rightarrow & \alpha\ B \\ p2: & B & \rightarrow & \beta \end{array}$$

- **-Rr[=nome_não–terminal]** Remove a recursividade à esquerda que se encontra no não–terminal *nome_não–terminal*. No seguinte exemplo, o slk transforma a gramática que está à esquerda naquela que se encontra à direita, removendo a recursividade à esquerda do símbolo A.

$$\begin{array}{lcl} A & \rightarrow & \alpha\ B\ |A\ \beta \\ B & \rightarrow & \varphi \end{array} \qquad \begin{array}{lcl} A & \rightarrow & \alpha\ B\ more\_A \\ more\_A & \rightarrow & \beta\ more\_A \mid \_epsilon\_ \\ B & \rightarrow & \beta \end{array}$$

  Para se obter o resultado, é necessário exportar a gramática gerada para um novo ficheiro, usando a opçõa '-f'

- **-Ru[=nome_não–terminal]** Faz remover as produções que não são usadas, isto é, aquelas que não são acessíveis a partir da produção inicial.

- **-Sn[=nome_não–terminal]** Faz a substituição de não–terminais por não–terminais que lhe sejam equivalentes.

- **-St[=nome_não–terminal]** Esta opção faz remover as produções inúteis de uma gramática, substituindo o não terminal *nome_não–terminal* pelo seu lado esquerdo, o qual deve ser sempre um único símbolo terminal.

- **-Tl=nome_ficheiro** Importa um ficheiro com tokens, para substituir a maneira automática como o slk os declara.

- **-Tw=nome_ficheiro** Exporta para um ficheiro todos os tokens que existem na gramática de entrada.

- **-t[=n]** Faz mostrar, passo a passo, todas as transformações que são produzidas na gramática. O n pode tomar valores entre 1 e 4 para se obter vários níveis para o trace efectuado às transformações da gramática.

- **-v[=n]** Faz com que o slk produza mais mensagens na fase de análise da gramática. Há três níveis para se ter mais ou menos mensagens durante esta fase: n=1 (equivalente a usar apenas '-v'), n=2 e n=3.

  Para a gramática colocada em exemplo para a opção '-Dt' se colocarmos '-v' com qualquer um dos níveis, vamos obter como mensagens a seguinte informação:

  ```
  SglInv: equivalentto id
  SglProj: equivalent to id
  Desc: equivalent to str
  ```

  Ou seja, que aqueles não terminais constituem produções inúteis, que podem portanto ser eliminados usando a opção '-St=[não–terminais]'

- **-y[s]** É usada quando a gramática de entrada está especificada em YACC. Deste deixa de ser apenas possível usar gramáticas escritas com a sintaxe do slk. A sub–opção 's' é usada sempre que a gramática do YACC está anotada com acções semânticas. Só assim a gramática é transformada correctamente para a sintaxe do slk.

  Esta opção é interessante pois passa a ser possível fazer uma análise métrica a gramáticas escritas em YACC.

3 Transformações das gramáticas EBNF

O slk suporta gramáticas especificadas em EBNF, contudo o parser gerado não utiliza a gramática especificada em EBNF. Portanto o slk realiza simples transformações à gramática de entrada de modo a especificá-la em BNF. Essas alterações são verificadas na adição de um novo símbolo não terminal e novas produções associadas a esse novo símbolo.

3.1 Zero ou mais: { símbolo }

Assim, quando na gramática temos uma produção como a produção em:

```
A : { B }
```

O slk vai transformar a gramática de modo a descrever as produções deste tipo em notação BNF[3].

A gramática vê–se então alterada em dois pontos, repare-se na gramática alterada:

```
A   : B_*
B_* : B B_*
B_* :
```

A produção inicial é alterada, é acrescentado um novo símbolo não terminal (B_*), e por fim são acrescentadas duas novas produções. Uma das produções é a derivação no vazio, ou seja, representa uma lista vazia. A outra produção representa uma lista com um ou mais símbolos B.

[3]O slk não suporta as extensões à notação BNF que normalmente são utilizados, assim o * equivale a { x }, + equivale a { x }+ e por fim, ? equivale a [x]

3.2 Um ou mais: { símbolo }+

Tal como na subsecção anterior, quando existem produções anotadas em EBNF, o slk vai efectuar transformações. Repare-se na gramática inicial:

```
A : { b }+
```

Desta gramática o slk vai efectuar a seguinte transformação:

```
A   : b b_*
b_* : b b_*
b_* :
```

Aqui o slk altera a produção inicial dizendo que A deriva em um símbolo 'b' e é seguido por uma continuação de b´s. Mas b_* é exactamente uma lista de zero ou mais símbolos 'b'. Deste modo, a gramática alterada reconhece exactamente a mesma linguagem., ou seja, uma lista de um ou mais símbolos 'b'.

3.3 Opcional: [símbolo]

Por último, quando queremos especificar que um dado símbolo é opcional, usamos a seguinte notação em EBNF suportado pelo slk:

```
A   : [ b ]
```

Contudo o slk, à semelhança das duas subsecções anteriores, vai alterar a gramática, colocando-a do seguinte modo:

```
A     : b_opt
b_opt :b
b_opt :
```

O slk altera a produção inicial tirando o caso de opção e adicionando um novo símbolo não–terminal (b_opt). Acrescenta ainda duas novas produções associadas ao novo símbolo terminal, onde uma delas deriva em vazio, e a outra deriva no símbolo 'b'. Ou seja, b_opt descreve exactamente que ou vai existir um símbolo 'b' ou não vai existir esse símbolo, portanto designa opção em ter ou não o símbolo b.

4 Usar o Slk em qualquer lugar

Quando se usa uma ferramenta que tem que ser invocada a partir da linha de comandos, muitas vezes é necessário estar na directoria onde o executável se encontra. Mas isto é bastante maçador no caso de a ferramenta se encontrar numa directoria cujo caminho em relação à raiz do sistema é bastante longo.

Para melhorar isso, e ter apenas que escrever esse longo caminho uma única vez, usam-se variáveis de ambiente que os sistemas operativos fornecem.

Imagine-se que queremos instalar a ferramenta slk na directoria C:\Programas\SLK, e, de qualquer ponto do sistema, a partir da linha de comandos, chamar apenas slk (que é o nome do executável).

Então, o que precisamos de fazer é seguir os seguintes passos:

1. Clicar com o botão direito do rato sobre "O meu computador" e escolher a opção **Propriedades**.

2. Escolher o separador **Avançadas**.

3. Clicar no botão **Variáveis de Ambiente**

4. Vão aparecer duas listagens de variáveis. Uma delas é **variáveis do sistema**. É essa que nos interessa. Clicar dias vezes sobre a variável de nome **PATH**.

5. Aparecerá uma janela com o nome da variável e uma lista de items (que são caminhos desde a raiz do sistema) separados pelo sinal ';'. Repare-se que o último não leva o sinal.

6. A seguir ao último item, coloca-se um ';' (sem as pelicas) e o caminho desde a raiz até à directoria onde ficará o slk. No caso deste exemplo ficaria:
   ...;C:\PROGRA~1\ SLK.

7. Clicar em OK nas três janelas abertas.

No final basta abrir a linha de comandos e invocar o slk sem escrever nada antes.

NOTA: O slk vai utilizar como directorias de output a directoria onde é invocado na linha de comandos. O seu input ou está na mesma directoria usada para o output, ou então é necessário escrever o caminho até ele.