

Compiladores

Análise sintática (5)
Gramáticas SLR(1), LR(1) e LALR
YACC

Análise Bottom-Up

- String Entrada -> Símbolo Inicial
 - Regras aplicadas em reverso
 - “adiar decisões” – mais poderoso
- Noção de handle, redução, uso de pilha
- Parsing **Shift-Reduce** (empilha-reduz)
 - LR(0)
 - SLR(1)
 - LR(1)
 - LALR(1)
- Exemplo:

$$\begin{aligned} S &\rightarrow T \\ T &\rightarrow F \mid T * F \\ F &\rightarrow \text{id} \mid (T) \end{aligned}$$

Parsing LR

- Parser baseado em tabelas**
 - Cria derivações mais a direita
- Usar um autômato finito**
 - Cada estado representa produções da gramática
 - Transições são feitas a cada terminal/não terminal
- Estruturas de dados:**
 - Pilha de estados {s}
 - Tabela de ações: Action[s,a]; $a \in T$
 - Tabela de transições: Goto[s,X]; $X \in N$

Tabela Ações/Transições

	*	(	)	id	\$	T	F
0		S5		S8		2	1
1	R1	R1	R1	R1	R1		
2	S3				Ok!		
3		S5		S8			4
4	R2	R2	R2	R2	R2		
5		S5		S8		6	1
6	S3		S7				
7	R4	R4	R4	R4	R4		
8	R3	R3	R3	R3	R3		

Análise LR

- switch (action[s, a])
 - caso shift u:
 - empilha estado u
 - ler novo a
 - caso reduzir r:
 - Procurar produção r: $X \rightarrow Y1..Yk$;
 - pop k estados, encontrar estado u
 - push goto[u,X]
 - caso aceitar: fim
 - case erro: erro

Construindo tabelas LR

- Configuração ou **ítem** (ou ítem LR(0))
 - = uma regra da gramática com um 'ponto' em algum lugar à direita.
- Exemplo: $T \rightarrow T * F$
- Possui quatro **ítems**:

$$\begin{aligned} T &\rightarrow \bullet T * F \\ T &\rightarrow T \bullet * F \\ T &\rightarrow T * \bullet F \\ T &\rightarrow T * F \bullet \end{aligned}$$
- O ponto • representa até onde foi feita a análise
- Calculá-se os conjuntos canônicos de ítems
 - = todos os ítems alcançáveis a partir de um conjunto de regras da gramática.
- Cada conjunto será um **estado** do parser

Cálculo dos conjuntos: Fechamento

Propriedade de Fechamento:

- Se $T \rightarrow X_1 \dots X_i \bullet X_{i+1} \dots X_n$ está em um conjunto, e X_{i+1} é um não-terminal que deriva em α , então também está no conjunto:

$$X_{i+1} \rightarrow \bullet \alpha$$

- Itera-se essa operação
 - Calcula o conjunto como um ponto fixo
- Adiciona-se um não-terminal S' à gramática
 - Adicione produção $S' \rightarrow S$
 - Conjunto de itens Inicial é:

$$\text{fechamento}(S' \rightarrow \bullet S)$$

Sucessor(C,X)

- Segundo procedimento útil
 - Pega em argumento um conjunto de itens C e um símbolo X
 - Retorna um conjunto de itens
 - Informalmente: "mover o ponto pelo símbolo X"
- 1. mover ponto para direita em todos os itens onde o ponto precede X
 - Para todas as regras $A \rightarrow \alpha \bullet X \beta$ em C retorna $A \rightarrow \alpha X \bullet \beta$
- 2. Calcular fechamento

Construção dos Conjuntos de Itens

- Família de conjuntos de itens
 - Cada conjunto será um **estado** do parser

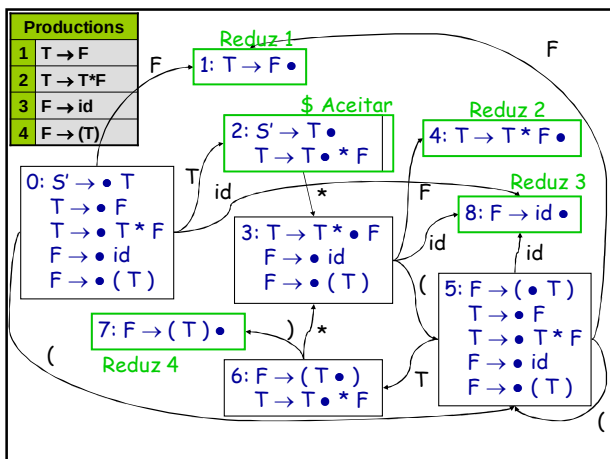
proc items(G')

```

C = fechamento({S' → • S});
do foreach I ∈ C do
    foreach X ∈ (N ∪ T) do
        C = C ∪ Sucessor(I, X);
while C é modificado;
```

Construção Tabela LR(0)

- Construir $F = \{I_0, I_1, \dots, I_n\}$
 - I_0 é o estado inicial
- Se $\{S' \rightarrow S \bullet\} \in I_i$ então ação[i,\$] = aceitar
 - Se $\{A \rightarrow \alpha \bullet\} \in I_i$ e $A \neq S'$ então ação[i,] = reduzir $A \rightarrow \alpha$
 - Se $\{A \rightarrow \alpha \bullet a \beta\} \in I_i$ e $\text{Sucessor}(I_i, a) = I_j$ então ação[i,a] = shift j
- Se $\text{Sucessor}(I_i, A) = I_j$ então goto[i,A] = j
- Todas as entradas não definidas são erros



Observações

- LR(0) sempre reduz se $\{A \rightarrow \alpha \bullet\} \in I_i$, sem lookahead
- Ítems Shift e Reduce podem estar no mesmo conjunto de configurações
 - Conflito Shift/Reduce
- Pode haver mais de um ítem reduce por conjunto
 - Conflito Reduce/Reduce
- Problema com $A \rightarrow \epsilon$

Exemplo de conflitos

$S' \rightarrow T$
 $T \rightarrow F \mid T * F$
 $F \rightarrow id \mid (T)$
 $F \rightarrow id = T$
 $T \rightarrow id$

5: $F \rightarrow id \bullet$
 $F \rightarrow id \bullet = T$

...

Conflito Shift/reduce!

2: $F \rightarrow id \bullet$
 $T \rightarrow id \bullet \dots$

Conflito Reduce/Reduce!

Uma solução "simples" para resolver as ambigüidades

- Em casos de gramáticas de **operadores**, pode-se usar uma **tabela de precedências**:
 - GLC sem produção ϵ
 - Não há dois não-terminais adjacentes nos lados direitos
 - Exemplo: $E \rightarrow E + E \mid E - E \mid E * E \mid (E) \mid id$
- Basta definir a **precedência** relativa entre os terminais
 - $* > +$ leia-se $**$ é prioritário sobre $+$
 - $(< *$ leia-se $($ é menos prioritário do que $*$
 - $id > +$
 - Etc...
 - Obtém-se uma tabela de comparação
- Percorre-se o texto de entrada até achar um símbolo com maior precedência;

Outra solução para resolver o problema

- Limitar as decisões de redução aos casos que o terminal que segue na entrada é compatível com a redução.
 - Look-ahead
- Isso pode tirar (alguns) conflitos shift/reduce.
- Não vai mudar o tamanho da tabela...
- Para saber quais terminais seguem um símbolo na entrada, usa-se
- É o método SLR(1)
 - Usa-se 1 token de look-ahead.

Outra solução para resolver o problema

- Limitar as decisões de redução aos casos que o terminal que segue na entrada é compatível com a redução.
 - Look-ahead
- Isso pode tirar (alguns) conflitos shift/reduce.
- Não vai mudar o tamanho da tabela...
- Para saber quais terminais seguem um símbolo na entrada, usa-se **Follow()**!
- É o método SLR(1)
 - Usa-se 1 token de look-ahead.
 - Simple LR Parser

Construção Tabela SLR(1)

- Construir $F = \{I_0, I_1, \dots, I_n\}$ (conjuntos de ítems LR(0))
 - I_0 é o estado inicial
- Se $\{S' \rightarrow S \bullet\} \in I_i$ então $ação[i, \$] = \text{aceitar}$
 - Se $\{A \rightarrow \alpha \bullet\} \in I_i$ e $A \neq S'$ então $ação[i, b] = \text{reduzir } A \rightarrow \alpha$ para todos $b \in \text{Follow}(A)$
 - Se $\{A \rightarrow \alpha \bullet a \beta\} \in I_i$ e $\text{Sucessor}(I_i, a) = I_j$ então $ação[i, a] = \text{shift } j$
- Se $\text{Sucessor}(I_i, A) = I_j$ então $\text{goto}[i, A] = j$
- Todas as entradas não definidas são erros

SLR(1) : Simple LR(1) Parser

0: $S' \rightarrow \bullet T$
 $T \rightarrow \bullet F$
 $T \rightarrow \bullet T * F$
 $T \rightarrow \bullet C(T)$
 $F \rightarrow \bullet id$
 $F \rightarrow \bullet id ++$
 $F \rightarrow \bullet (T)$
 $C \rightarrow \bullet id$

$S' \rightarrow T$
 $T \rightarrow F \mid T * F \mid C(T)$
 $F \rightarrow id \mid id ++ \mid (T)$
 $C \rightarrow id$

id

1: $F \rightarrow id \bullet$
 $F \rightarrow id \bullet ++$
 $C \rightarrow id \bullet$

$\text{Follow}(F) = \{*,), \$\}$
 $\text{Follow}(C) = \{ (\}$

$ação[1, *] = ação[1, \$] = ação[1, \$] =$
 $ação[1, (] =$
 $ação[1, +] =$

$\text{Reduz } F \rightarrow id$
 $\text{Reduz } C \rightarrow id$
 Shift

Observações sobre SLR(1)

- Obs: SLR(1) somente reduz $\{A \rightarrow \alpha \bullet\}$ se o lookahead está em $\text{Follow}(A)$
- Ítens empilha e reduz podem estar no mesmo conjunto de configuração desde que os seus "lookaheads" sejam disjuntos
- Uma gramática é SLR(1) se para cada conjunto de configuração:
 - Para cada item $\{A \rightarrow \alpha \bullet a\beta : a \in T\}$:
não existe nenhum $\{B \rightarrow \gamma \bullet : a \in \text{Follow}(B)\}$
IMPEDE conflito Reduz/shift
 - Para cada dois itens $\{A \rightarrow \gamma \bullet\}$ e $\{B \rightarrow \beta \bullet\}$ $\text{Follow}(A) \cap \text{Follow}(B) = \emptyset$
IMPEDE conflito Reduz/Reduz

Gramáticas LR(0) $\subset$ Gramáticas SLR(1)

Exercício SLR(1)

- Considere a seguinte gramática livre do contexto:

$S \rightarrow a(L) \mid a$
 $L \rightarrow S, L \mid S$

- Construir os estados do parser SLR(1)
- Calcular os conjuntos first e follow da gramática
- Calcular a tabela SLR(1)
- Mostrar análise de $a(a,a)$

Sumário: análise sintática

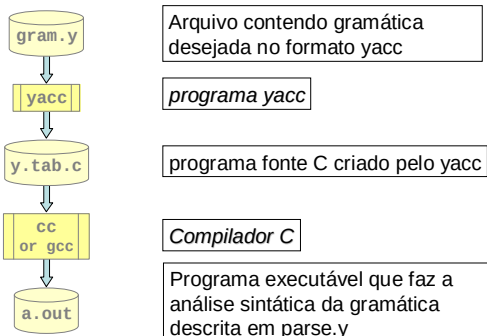
- Top-Down - recursiva ou com tabela preditiva LL(1):
 - Fácil de implementar
 - Necessita apenas o First/Follow
 - Incompatível com GLCs recursivas a esquerda.
- Bottom-up – LR(0), LR(1), SLR(1), LALR(1)...
 - Mais poderosas (= reconhecem mais GLCs)
 - Implementações muito eficientes
 - Mais difíceis de implementar à mão.

YACC

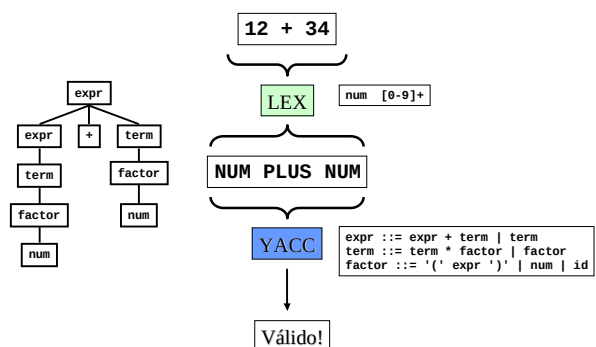
- Yet Another Compiler Compiler
- Produz um *parser* bottom-up para uma dada gramática
- Usado para produzir compiladores para Pascal, C, C++ entre outras
- Além disso, foi usado no desenvolvimento de:
 - bc - calculadora
 - eqn & pic
 - verificador de sintaxe SQL
 - Lex
- **bison**: substituto do yacc da GNU

YACC

Seqüência básica operacional



Exemplo



Formato do arquivo YACC

Definições

%%

Regras

%%

Código Suplementar

Seções de Regras

- Normalmente escritas como segue:

```
expr  : expr '+' term
      | term
      ;
term   : term '*' factor
      | factor
      ;
factor : '(' expr ')'
      | ID
      | NUM
      ;
```

Seção de Definições

```
%{
#include <stdio.h>
#include <stdlib.h>
%}
%token ID NUM
%start expr
```

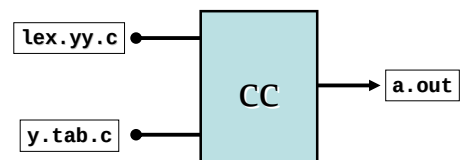
Obs:

- lex produz uma função **yylex()**
- yacc produz uma função **yyparse()**
- yyparse espera chamar uma yylex
- **Como conseguir yylex?**
 - Escrever sua própria!
 - Usar Lex

Construindo yylex

```
int yylex()
{
    if(it's a num)
        return NUM;
    else if(it's an id)
        return ID;
    else if(parsing is done)
        return 0;
    else if(it's an error)
        return -1;
}
```

lex & yacc



Exemplo

- Suponha um arquivo lex **scanner.l** e um arquivo yacc chamado **decl.y**.
- Passos a serem feitos ...

```
yacc -d decl.y
lex scanner.l
gcc -c lex.yy.c y.tab.c
gcc -o parser lex.yy.o y.tab.o -ll
```

Nota: scanner deve incluir na seção de definições:

```
#include "y.tab.h"
```

YACC

- As regras podem ser recursivas
- As regras não podem ser ambíguas*
- Usa um parser bottom up Shift/Reduce -LALR(1)
 - Solicita um token
 - Empilha
 - Redução ?
 - Sim: reduz usando a regras correspondente
 - Não: pega outro token
- Yacc não pode olhar mais que um token de lookahead
- yacc -v gram.y gera a tabela de estados, em y.output

Exemplo: Lex

scanner.l

```
%{
#include <stdio.h>
#include "y.tab.h"
}%
id      [_a-zA-Z][_a-zA-Z0-9]*
wspc    [ \t\n]+
semi    [;]
comma   [,]
%%
int      { return INT; }
char     { return CHAR; }
float    { return FLOAT; }
{comma}  { return COMMA; }
{semi}   { return SEMI; }
{id}     { return ID; }
{wspc}   {;}
```

Exemplo: Definições

decl.y

```
%{
#include <stdio.h>
#include <stdlib.h>
}%
%start line
%token CHAR, COMMA, FLOAT, ID, INT, SEMI
%%
```

Exemplo: Regras

decl.y

```
decl : type ID list
      { printf("Success!\n");
      } ;
list : COMMA ID list
      | SEMI
      ;
type : INT | CHAR | FLOAT
      ;
%%
```

Testando

```
yacc -d decl.y
```

- Produced **y.tab.h**

```
# define CHAR 257
# define COMMA 258
# define FLOAT 259
# define ID 260
# define INT 261
# define SEMI 262
```