Compiladores (CC3001)
Aula 7: Geradores de analisadores sintáticos

Mário Florido

DCC/FCUP

2024

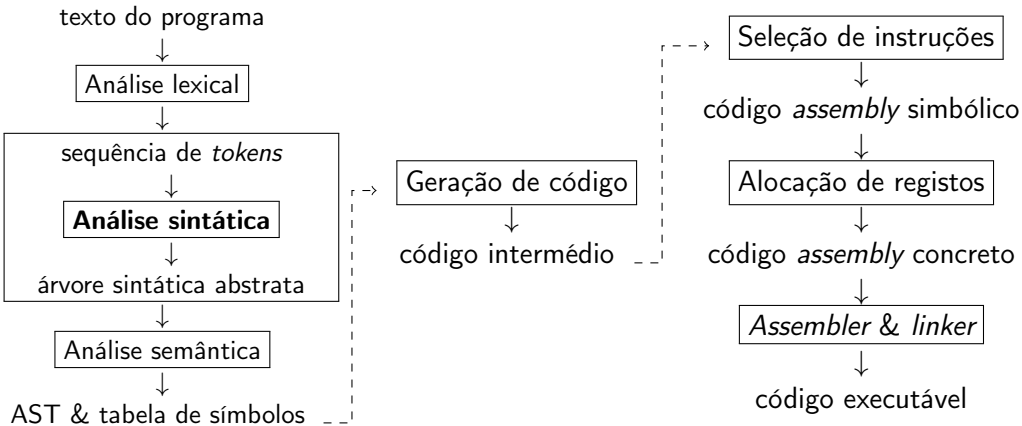
[dcc] FC FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Esta aula

Geradores de analisadores sintáticos
- *Happy*
- *Bison*
- Conflitos

Sintaxe abstrata
- Sintaxe concreta e abstrata
- Linguagens funcionais
- Linguagem C
- Java e linguagens OO
- Outras

Re-lembrar: fases dum compilador

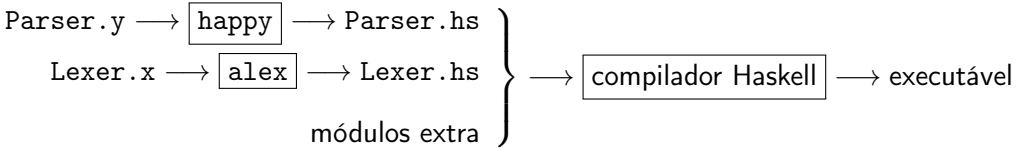


- Na aula passada: vimos que o análisador LR é um autómato com transições definidas por uma tabela
- Este autómato pode ser gerado automaticamente por um programa:

Yacc	"Yet Another Compiler Compiler" gerador de analisadores LALR do sistema UNIX (para a linguagem C)
Bison	uma re-implementação GNU do Yacc (para C ou C++)
Happy	gerador semelhante ao Yacc e Bison que produz analisadores LALR em Haskell

- Existem também geradores automáticos para analisadores LL (mas não vamos usar)
- Nesta aula: vamos ver o *Happy* e *Bison* (e também estruturas de dados para ASTs)

- Um gerador de analisadores sintáticos LALR(1) para Haskell
- Recebe um ficheiro com a gramática da linguagem anotada
  - produções anotadas com ações semânticas (expressões Haskell)
  - extras: declarações, diretivas e código auxiliar
- Concebido para integrar um analisador lexical gerado pelo *Alex*
- Gera automaticamente o código do analisador sintático




```
$ alex Lexer.x
$ happy Parser.y
$ ghc -o prog Lexer.hs Parser.hs ...
```

Exemplo

```
1  {
2  module Parser where
3  import Lexer
4  }
5
6  %name parser
7  %tokentype { Token }
8  %error { parseError }
9
10 %token
11
12 num         { TOK_NUM $$ }
13 '+'         { TOK_PLUS }
14 '('         { TOK_LPAREN }
15 ')'         { TOK_RPAREN }
16 %%
17
18 Exp : Term                 { () }
19     | Exp '+' Term         { () }
20
21 Term : num                 { () }
22      | '(' Exp ')'         { () }
23
24 {
25 parseError :: [Token] -> a
26 parseError toks = error "parse error"
27 }
```

- O tipo de *tokens* está definido no Lexer:

  ```
  data Token = TOK_NUM Int | TOK_PLUS | TOK_LPAREN | TOK_RPAREN
  ```

- Diretiva `%name` define o nome da função de *parsing*
- O primeiro não-terminal da gramática é inicial (no exemplo: `Exp`)
- As ações semânticas entre chavetas são expressões Haskell
- No exemplo: todas as produções retornam `()`
- Logo: a função de *parsing* gerada pelo Happy terá tipo

  ```
  parser :: [Token] -> ()
  ```

  (não retorna nenhum resultado útil.)

Exemplo (cont.)

```
module Main where
import Parser
import Lexer

main :: IO ()
main = do
   txt <- getContents
   print (parser $ alexScanTokens txt)
```

Este programa lê toda a entrada padrão e verifica se respeita a gramática.

- se sim: imprime `()`;
- caso contrário: lança uma exceção.

Ações semânticas

- Além de reconher uma linguagem

  um compilador deve construir uma representação do programa (e.g. árvore sintática)

  um interpretador deve executar o programa
- Podemos implementar estas extensões acrescentado ações semânticas apropriadas a cada produção da gramática

Construir a árvore sintática


```
{
module Parser where
import Lexer
}

%name parser
%tokentype { Token }
%error { parseError }

%token

num         { TOK_NUM $$ }
'+'         { TOK_PLUS }
'('         { TOK_LPAREN }
')'         { TOK_RPAREN }
%%

Exp : Term                  { $1 }
    | Exp '+' Term          { Add $1 $3 }

Term : num                  { Num $1 }
     | '(' Exp ')'          { $2 }

{
data Exp = Num Int
         | Add Exp Exp
         deriving Show

parseError :: [Token] -> a
parseError toks = error "parse error"
}
```

Construir a árvore sintática (cont.)

- Declaramos um novo tipo algébrico para árvores sintáticas:

  ```
  data Exp = Num Int | Add Exp Exp | ...
  ```

- Acrescentamos uma ação a cada produção que constroi uma árvore a partir das sub-árvores:
  - $1, $2, etc. referirem os valores semânticos dos terminais e não-terminais
  - os valores de não-terminais são árvores `Exp`
  - os valores de terminais são definidos na secção `%token`

    ```
    num          { TOK_NUM $$ }
    ```

    (e.g. o valor de `TOK_NUM 42` é o inteiro 42)

- A função de *parsing* gerada terá tipo

  ```
  parser :: [Token] -> Exp
  ```

(Ver demonstração.)

Calcular o resultado

- Em vez de construir uma árvore podemos calcular imediatamente o valor de cada expressão
- O valores semântico de cada ação vai ser `Int`
- A função de *parsing* gerada pelo Happy passa a

  ```
  parser :: [Token] -> Int
  ```
- A função `parse` implementa um interpretador: recebe a sequência de *tokens* e calcula o valor da expressão (ou lança uma exceção)

Calcular o resultado (cont.)


```
{
module Parser where
import Lexer
}

%name parser
%tokentype { Token }
%error { parseError }
⋮
```

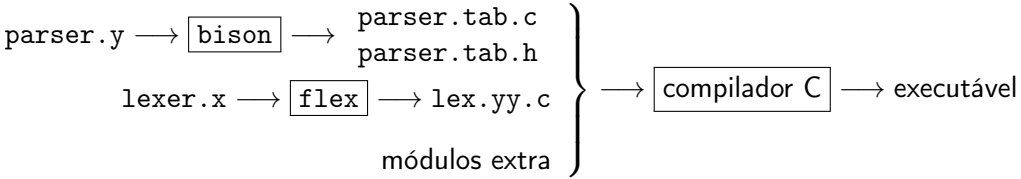
```
%%

Exp : Term                { $1 }
    | Exp '+' Term        { $1 + $3 }

Term : num                { $1 }
     | '(' Exp ')'        { $2 }

{
parseError :: [Token] -> a
parseError toks = error "parse error"
}
```

- Gerador de analisadores LR para C e C++ do projeto GNU
- Semelhante ao Yacc (por transitividade também ao *Happy*)
- Concebido para integrar um analisador lexical gerado pelo *Flex*




```
$ bison -d parser.y
$ flex lexer.x
$ gcc -o prog parser.tab.c lex.yy.c ...
```

```
%{
#include <stdio.h>
#include <stdlib.h>
int yylex (void);
void yyerror (char const *);
%}
%define api.value.type {int}
%token TOK_NUM

%%

top : exp      { printf("%d\n", $1); }
    ;

exp :  term          { $$ = $1; }
    | exp '+' term   { $$ = $1 + $3; }
    ;

term : TOK_NUM       { $$ = $1; }
     |  '(' exp ')'  { $$ = $2; }
     ;

%%

void yyerror (char const *msg) {
  printf("parse error: %s\n", msg);
  exit(-1);
}

int main(void) {
  yyparse();
}
```

- Cada ação é um bloco de código em C usando `$1`, `$2`, etc.
- Para definir o valor do resultado `$$` usamos uma atribuição (em vez de uma expressão no *Happy*)
- O tipo de valores semânticos é definido por

  `%define api.value.type {int}`

  (por omissão é `int`)
- O valor dos terminais deve ser colocado pelo analisador lexical em `yylval` (global)

O *Happy* e *Bison* reportam conflitos se a gramática for ambígua.[1]

Parser.y

```
⋮
Exp : num                    { ... }
    | Exp '+' Exp            { ... }
    | Exp '*' Exp            { ... }
⋮
```

```
$ happy Parser.y
shift/reduce conflicts:  4
```

Conflitos (cont.)

- O *Happy* e *Bison* permitem gerar relatórios dos estados do autómato LR e eventuais conflitos

```
$ happy -i Parser.y          # produz Parser.info
$ bison --report parser.y    # produz parser.output
```

- Por omissão: os conflitos são resolvidos fazendo *shift*
- Dependendo da gramática isto pode não ser a opção correta!
- Por vezes devemos re-escrever a gramática para remover ambiguidades
- Alternativa: especificar associatividade e precedência de *tokens*
  (estas declarações são idênticas em *Happy* e *Bison*; veremos exemplos do primeiro)

[1] Ou apenas não for $LALR(1)$.

- Em vez de re-escrever a gramática podemos declarar a precedências e associatividades dos *tokens*
- Associatividades podem ser: `%left`, `%right`, `%nonassoc`
- As prioridade são dadas pela ordem das declarações

```
%nonassoc '<' '>' '=='          precedência mais baixa
%left '+' '-'
%left '*' '/'                   precedência mais elevada
```

A associatividade permitem resolver ambiguidades que envolvem um só operador.
Exemplo: 1+2+3.

```
Exp -> Exp . '+' Exp                    (rule 2)
Exp -> Exp '+' Exp .                    (rule 2)

'+'             shift, and enter state 6
                 (reduce using rule 2)
```

- Declarando `%left '+'` fazemos *reduce*; 1+2+3 analisado como $(1 + 2) + 3$
- Se fosse `%right '+'`: fazemos *shift*; 1+2+3 analisado como $1 + (2 + 3)$
- Se fosse `%nonassoc '+'`: 1+2+3 é um erro sintático

A ordem de precedência permite resolver ambiguidades entre operadores.

Exemplo: 1+2*3.

```
Exp -> Exp . '+' Exp                    (rule 2)
Exp -> Exp '+' Exp .                    (rule 2)
Exp -> Exp . '*' Exp                    (rule 3)

'*'             shift, and enter state 7
                 (reduce using rule 2)
```

- Como a precedência de $*$ é maior que a de $+$ optamos por *shift*
- Logo: 1+2*3 é analisado como $1 + (2 \times 3)$.

- Nem sempre é necessário eliminar o conflito *shift/reduce*
- Por omissão: os geradores LR escolhem *shift*
- Isto é a resolução correta para o *dangling else* (aula anterior)

```
Stm -> if Exp then Stm .                          (rule 1)
Stm -> if Exp then Stm . else Stm                 (rule 2)

else            shift, and enter state 15
                 (reduce using rule 1)
```

Happy `https://www.haskell.org/happy/doc/html/index.html`

Bison `https://www.gnu.org/software/bison/manual/html_node/index.html`

A árvore de derivação pode ter detalhes que não são relevantes para o resto do compilador:

- parêntesis usados para agrupar sub-expressões
- combinações de palavras reservadas para delimitar blocos (e.g. if/then/else ou begin/end)
- símbolos não-terminais usados para desambiguar a gramática

Podemos aproveitar a análise sintática para remover detalhes construindo uma árvore sintática abstrata (AST).

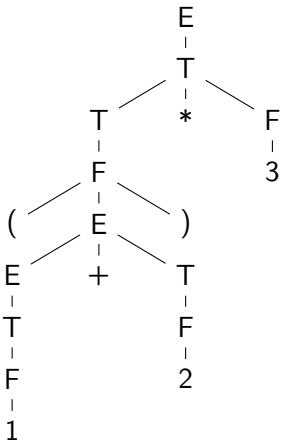
Exemplo

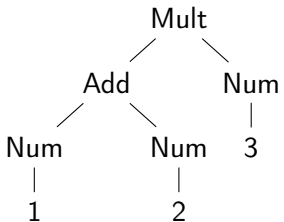
Considere a gramática de expressões

$$\begin{array}{lll} E \to E + T & T \to T * F & F \to \text{num} \\ E \to T & T \to F & F \to (\ E\ ) \end{array}$$

e a derivação

$$E \Rightarrow T \Rightarrow T * F \Rightarrow F * F \Rightarrow (E) * F \Rightarrow \cdots \Rightarrow \texttt{(1+2)*3}$$





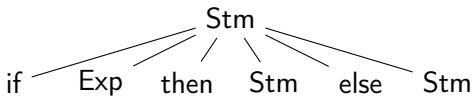
A árvore sintática concreta (à esquerda) contém muita informação redundante.
A árvore sintática abstrata (à direita) contém apenas um nó por cada operação.

Exemplo (cont.)

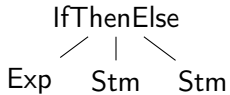
Outro exemplo: um fragmento de gramática para comandos.

$$Stm \rightarrow \text{if } Exp \text{ then } Stm \text{ else } Stm$$
$$Stm \rightarrow etc.$$

Na árvore sintática concreta de cada palavra reservada é um nó separado:



Podemos agrupar if/then/else na árvore sintática abstrata como um só nó:



Vantagens da sintaxe abstrata

- Simplifica as fases posteriores do compilador
  (evita a necessidade de considerar todos os detalhes da gramática)
- Permite análise semântica e geração de código sejam menos acopladas à linguagem-fonte
- Facilita a implementação de construções da linguagem-fonte como "*açucar sintático*"
  - exemplo: podemos re-escrever

$$x{>}y \quad \longrightarrow \quad y{<}x$$
$$x{>}{=}y \quad \longrightarrow \quad y{<}{=}x$$

    e só necessitamos de implementar 2 em vez de 4 operadores de comparação
  - `https://en.wikipedia.org/wiki/Syntactic_sugar`

- O compilador tem de construir e processar árvores de sintaxe abstrata (*ASTs*)
- Vamos ver como fazer em algumas linguagens de programação

Em ML, OCaml, Haskell ou F# podemos representar cada categoria da AST como um tipo algébrico:

- Uma enumeração de alternativas etiquetadas por construtores
- Cada construtor pode ter atributos de tipos diferentes e em número diferente
- As alternativas podem ser recursivas (direta ou indiretamente)
- Para discriminar alternarivas usamos encaixe de padrões

```
data Stm = AssignStm String Exp  -- ident = exp
         | IncrStm String        -- ident++
         | CompoundStm Stm Stm   -- stm1; stm2

data Exp = IdExp String          -- x, y, z, etc.
         | NumExp Int            -- 123, etc.
         | OpExp Exp BinOp Exp   -- e1+e2, e1*e2, ...

data BinOp = Plus | Minus | Times | Div
```

Exemplo de construção de uma AST:

```
example :: Stm
example =
  CompoundStm
  (AssignStm "a"
    (OpExp (NumExp 5) Plus (NumExp 3))
  )
  (AssignStm "b"
    (OpExp (IdExp "a") Minus (NumExp 2))
  )
```

Encaixe de padrões usando equações:

```
process :: Stm -> ...
process (IncrStm id) = ...
process (AssignStm id exp) = ...
process (CompoundStm s1 s2) = ...
```

Ou usando case:

```
process stm = case stm of
   IncrStm id -> ...
   AssignStm id exp -> ...
   CompoundStm s1 s2 -> ...
```

- Uma estrutura para cada categoria sintática:
  - uma etiqueta (*tag*);
  - uma união de alternativas;
  - cada alternativa será uma estrutura se tiver mais do que um atributo
- Funções auxiliares para construir os nós e inicializar todos os campos
- Programação seguindo um estilo funcional:
  - estruturas imutáveis — inicializadas pelos construtores e nunca modificadas
  - memória deve ser libertada apenas no final (duma fase ou do programa)
- Para discriminar alternativas: usamos a *tag*

Exemplo

```
struct _stm {
  enum
   {COMPOUND, ASSIGN, INCR} tag;
  union {
    struct {     // for COMPOUND
      struct _stm *fst, *snd;
    } compound;
    struct {       // for ASSIGN
      char *ident;
      struct _exp *expr;
    } assign;
    char *ident;   // for INCR
  };
};

typedef struct _stm *Stm;
```

```
typedef enum {PLUS, MINUS, TIMES, DIV} binop;
struct _exp {
  enum {ID, NUM, OP} tag;
  union {
    int val;           // for NUM
    char *id;          // for ID
    struct {           // for OP
      binop op;
      struct _exp *left, *right;
    } binop;
  };
};
typedef struct _exp *Exp;
```

Exemplo (cont.)


```
Exp mk_num(int v) {
  Exp e = (Exp) malloc(sizeof(struct _exp));
  e->tag = NUM;
  e->val = v;
  return e;
}

Exp mk_ident(char *txt) {
  Exp e = (Exp)malloc(sizeof(struct _exp));
  char *str = malloc(strlen(txt)+1);
  strcpy(str, txt);   // é preferível ‘‘clonar’’ a cadeia
  e->tag = ID;
  e->id = str;
  return e;
}
```

Exemplo (cont.)


```
Exp mk_op(binop op, Exp e1, Exp e2) {
  Exp e = (Exp) malloc(sizeof(struct _exp));
  e->tag = OP;
  e->binop.op = op;
  e->binop.left = e1;
  e->binop.right = e2;
  return e;
}
```

(Funções similares para outros construtores.)

Exemplo de construção de uma AST:

```
Stm example =
    mk_compound
    (mk_assign("a", mk_op(PLUS, mk_num(5), mk_num(3))),
     mk_assign("b", mk_op(TIMES, mk_ident("a"), mk_num(2))));
```

Exemplo (cont.)

Exemplo de uso em ações semânticas de *Yacc*/*Bison*:

```
exp : term            { $$ = $1; }
    | exp '+' term    { $$ = mk_binop(PLUS, $1, $3); }
    ;

term : TOK_NUM        { $$ = mk_num($1); }
     |  '(' exp ')'   { $$ = $2; }
     ;
```

Exemplo de análise de casos:

```
void process(Stm stm) {
   switch(stm->tag) {
     case COMPOUND: // usar stm->compound.fst e stm->compound.snd
     ⋮
     break;
     case ASSIGN:   // usar stm->assign.ident e e stm->assign.expr
     ⋮
     break;
     case INCR: // etc.
     ⋮
     break;
   }
}
```

Em Java (e outras linguagens OO):

- Uma classe abstrata para cada categoria sintática;
- Uma subclasse para cada alternativa com os atributos
- Cada subclasse tem um construtor que inicializa todos os atributos
- As estruturas devem ser imutáveis: inicializadas pelos construtores e nunca modificadas
- Para discriminar alternativas: usamos o operador `instanceof` e *downcasts* (alternativa: visitor design pattern)

Exemplo

```
public abstract class Stm {}
```

```
public class CompoundStm extends Stm
{
  public Stm fst, snd;
  public CompoundStm(Stm s1, Stm s2)
  {
     fst = s1; snd = s2;
  }
}
```

```
public class AssignStm extends Stm
{
  public String id;
  public Exp exp;
  public AssignStm(String i, Exp e)
  {
    id=i; exp=e;
  }
}
```

(Outras subclasses para restantes alternativas.)

```
public abstract class Exp {}
```

```
public class IdExp extends Exp {
  public String id;
  public IdExp(String i) {
    id=i;
  }
}
```

```
public class NumExp extends Exp {
  public int num;
  public NumExp(int n) {
    num=n;
  }
}
```

```
public class OpExp extends Exp {
  public Exp left, right;
  public int oper;
  final public static int Plus=1, Minus=2, Times=3, Div=4;
  public OpExp(Exp e1, int op, Exp e2) {
    left=e1;  oper=o; right=e2;
  }
}
```

Exemplo (cont.)


```
Stm prog =
   new CompoundStm(new AssignStm("a",
                   new OpExp(new NumExp(5),
                             OpExp.Plus, new NumExp(3))),
                   new AssignStm("b",
                   new OpExp (new IdExp("a"),
                              OpExp.Times, new NumExp(2))));
```

Exemplo (cont.)

Exemplo de análise de casos:

```
public void process(Stm stm) {
  if(stm instanceof CompoundStm) {
      CompoundStm cstm = (CompoundStm)stm;
      ⋮
  }
  else if(stm instanceof AssignStm) {
       AssignStm astm = (AssignStm)stm;
       ⋮
  }
  else ...
}
```

- Linguagens multi-paradigma frequentemente suportam tipos algébricos e encaixe de padrões ao estilo funcional; e.g.:
  - "case classes" em Scala
  - "data classes" em Kotlin
  - "enumerations" em Swift e Rust
- Para implementar ASTs: preferir a abordagem funcional em vez da OO