TÉCNICO LISBOA

Compiladores[1]

Avaliação dirigida pela sintaxe. Bison/Yacc

Capítulo 5 - "Compilers: Principles, Techniques and Tools"

Prof. Alberto Abad

IST - Universidade de Lisboa

2021/2022

[1] Slides adaptados de Prof. Pedro T. Monteiro (2017/2018)

Outline

Avaliação dirigida pela sintaxe

O resultado do analisador sintáctico é:

- TRUE - se o programa estiver correcto sintacticamente
- FALSE - se o programa não estiver correcto sintacticamente

As regras sintácticas têm associadas acções, que são guiadas pela estrutura sintáctica (*syntax-directed translation*).

Tarefas:

- Inserir informações na tabela de símbolos
- Efectuar verificações semânticas (e.g., tipos das variáveis e expressões)
- Gerar código intermédio, para posteriormente gerar código-máquina
- Emitir mensagens de erro
- Avaliar expressões

Vantagens:

- Maior rapidez – efectuada uma única passagem sobre a sequência de entrada
- Menos recursos memória – não é necessária a construção de uma representação do programa

Outline

Avaliação dirigida pela sintaxe é composta por:

- Gramática livre de contexto G
- Conjunto de atributos associados aos símbolos de G
- Conjunto de acções semânticas associadas às produções de G

As acções semânticas são executadas quando a produção é seleccionada pelo analisador sintáctico.

As acções semânticas são executadas quando a produção é seleccionada pelo analisador sintáctico.

Exemplo:

Produção	Regra semântica
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \rightarrow T$	$E.val = T.val$
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow (E)$	$F.val = E.val$
$F \rightarrow \mathbf{num}$	$F.val = \mathbf{num}.lexval$

A cada produção $A \rightarrow \alpha_1 \alpha_2 \ldots \alpha_n$, é associado um conjunto de regras semânticas $b = f(c_1, c_2, \ldots, c_k)$, onde f é uma função, e b e c_i são atributos.

Gramáticas atributivas

Atributos

A cada produção $A \rightarrow \alpha_1 \alpha_2 \ldots \alpha_n$, é associado um conjunto de regras semânticas $b = f(c_1, c_2, \ldots, c_k)$, onde f é uma função, e b e c_i são atributos.

O atributo b, associado a um **símbolo não terminal**, é considerado:

- **sintetizado**
  - b é associado a A
  - c_i são atributos associados aos símbolos α_j

A cada produção $A \rightarrow \alpha_1 \alpha_2 \ldots \alpha_n$, é associado um conjunto de regras semânticas $b = f(c_1, c_2, \ldots, c_k)$, onde f é uma função, e b e c_i são atributos.

O atributo b, associado a um **símbolo não terminal**, é considerado:

- **sintetizado**
  - b é associado a A
  - c_i são atributos associados aos símbolos α_j
- **herdado**
  - b é associado a um α_t
  - c_i é associado a A e/ou aos símbolos α_j $(j \neq t)$

A cada produção $A \rightarrow \alpha_1 \alpha_2 \ldots \alpha_n$, é associado um conjunto de regras semânticas $b = f(c_1, c_2, \ldots, c_k)$, onde f é uma função, e b e c_i são atributos.

O atributo b, associado a um **símbolo não terminal**, é considerado:

- **sintetizado**
  - b é associado a A
  - c_i são atributos associados aos símbolos α_j
- **herdado**
  - b é associado a um α_t
  - c_i é associado a A e/ou aos símbolos α_j $(j \neq t)$

Nota:
Atributos associados a **símbolos terminais**, são sempre **sintetizados**.
São calculados durante a análise lexical.

Exemplo com atributos sintetizados:

Produção	Regra semântica
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow \textbf{digit}$	$F.val = \textbf{digit}.lexval$

Exemplo com atributos sintetizados:

Produção	Regra semântica
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
$T \rightarrow F$	$T.val = F.val$
$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

Exemplo com atributos sintetizados e herdados:

Produção	Regra semântica
$T \rightarrow FT'$	$T'.inh = F.syn; T.syn = T'.syn$
$T' \rightarrow *FT_1'$	$T_1'.inh = T'.inh * F.syn; T'.syn = T_1'.syn$
$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
$F \rightarrow$ **digit**	$F.syn =$ **digit**$.lexval$

Outline

Grafos de dependências

Uma definição semântica pode conter simultâneamente vários atributos herdados e sintetizados.

Grafos de dependências

Uma definição semântica pode conter simultâneamente vários atributos herdados e sintetizados.

Necessário: ordenar as acções de forma a que as avaliações sejam feitas pela ordem correcta!

Grafos de dependências

Uma definição semântica pode conter simultâneamente vários atributos herdados e sintetizados.

Necessário: ordenar as acções de forma a que as avaliações sejam feitas pela ordem correcta!

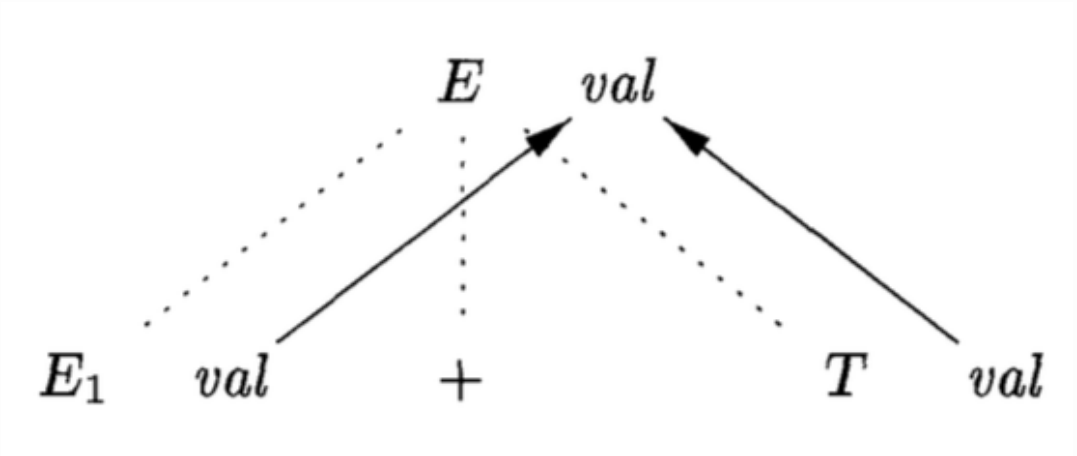
Algoritmo:

- Nó para cada atributo associado a símbolo X
- Se atributo sintetizado $A.b$ é definido em termos de $X.c$ adicionar arco de $X.c$ para $A.b$
- Se atributo herdado $B.c$ é definido em termos de $X.a$ adicionar arco de $X.a$ para $B.c$

Grafos de dependências

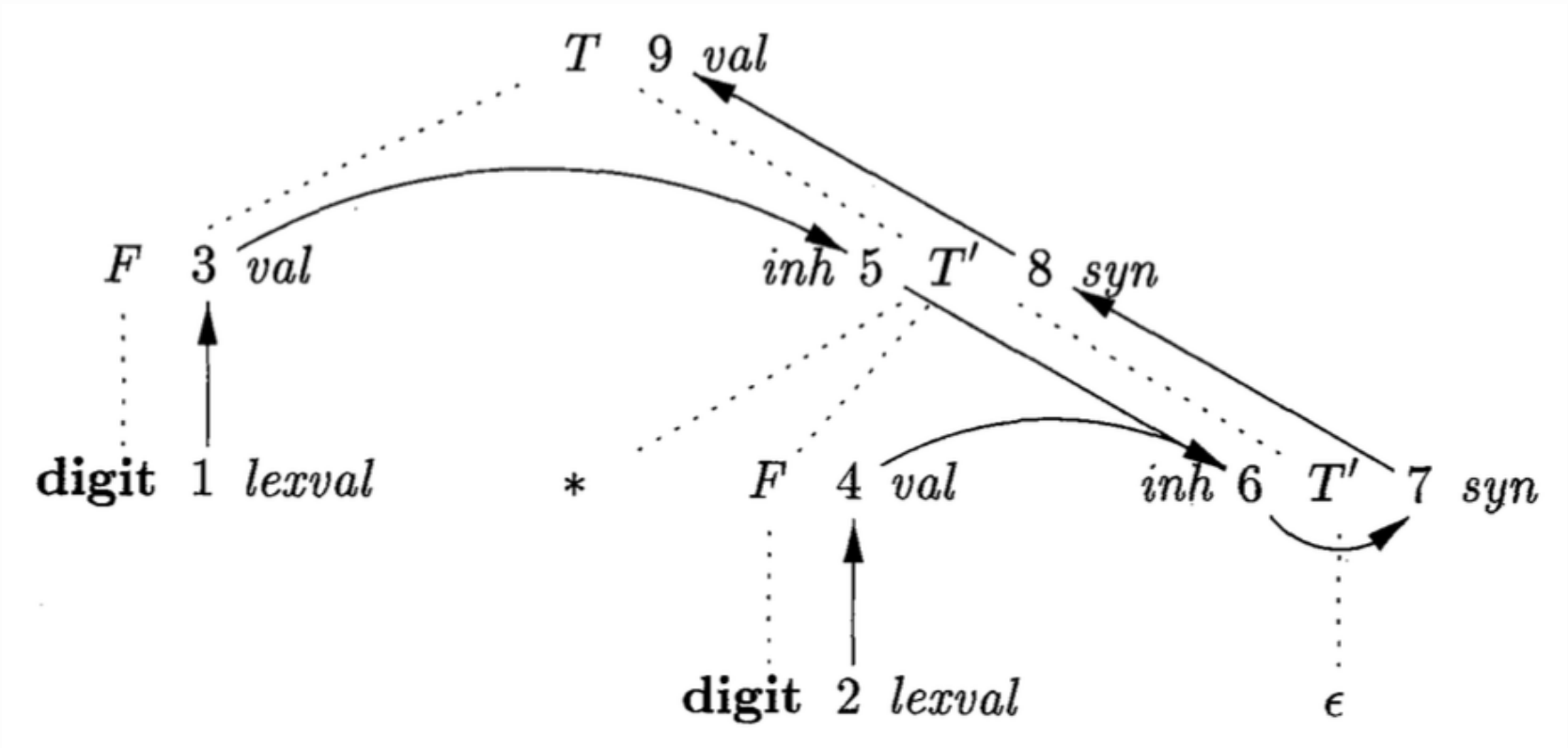
Exemplo 1

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 \ + \ T$	$E.val = E_1.val + T.val$



Grafos de dependências

Exemplo 2



Gramáticas atributivas

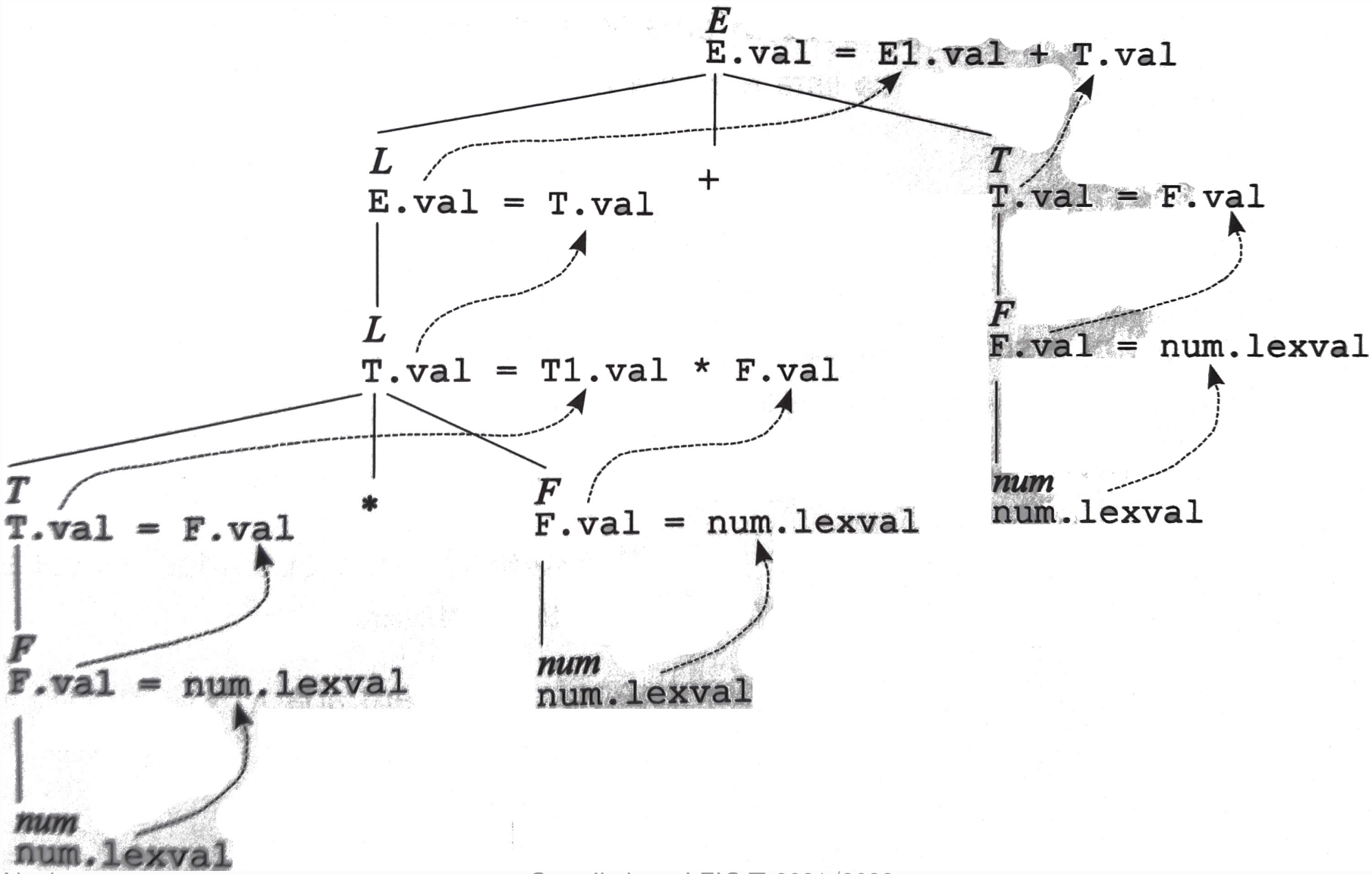
Gramática de S-atributos:

- Os valores dos atributos dependem apenas dos valores dos atributos dos nós filhos
- Atributos são avaliados de forma ascendente

Pode ser usado na análise sintáctica ascendente (Parsing LR).

Grafo de dependências

Exemplo – Gramática de S-atributos



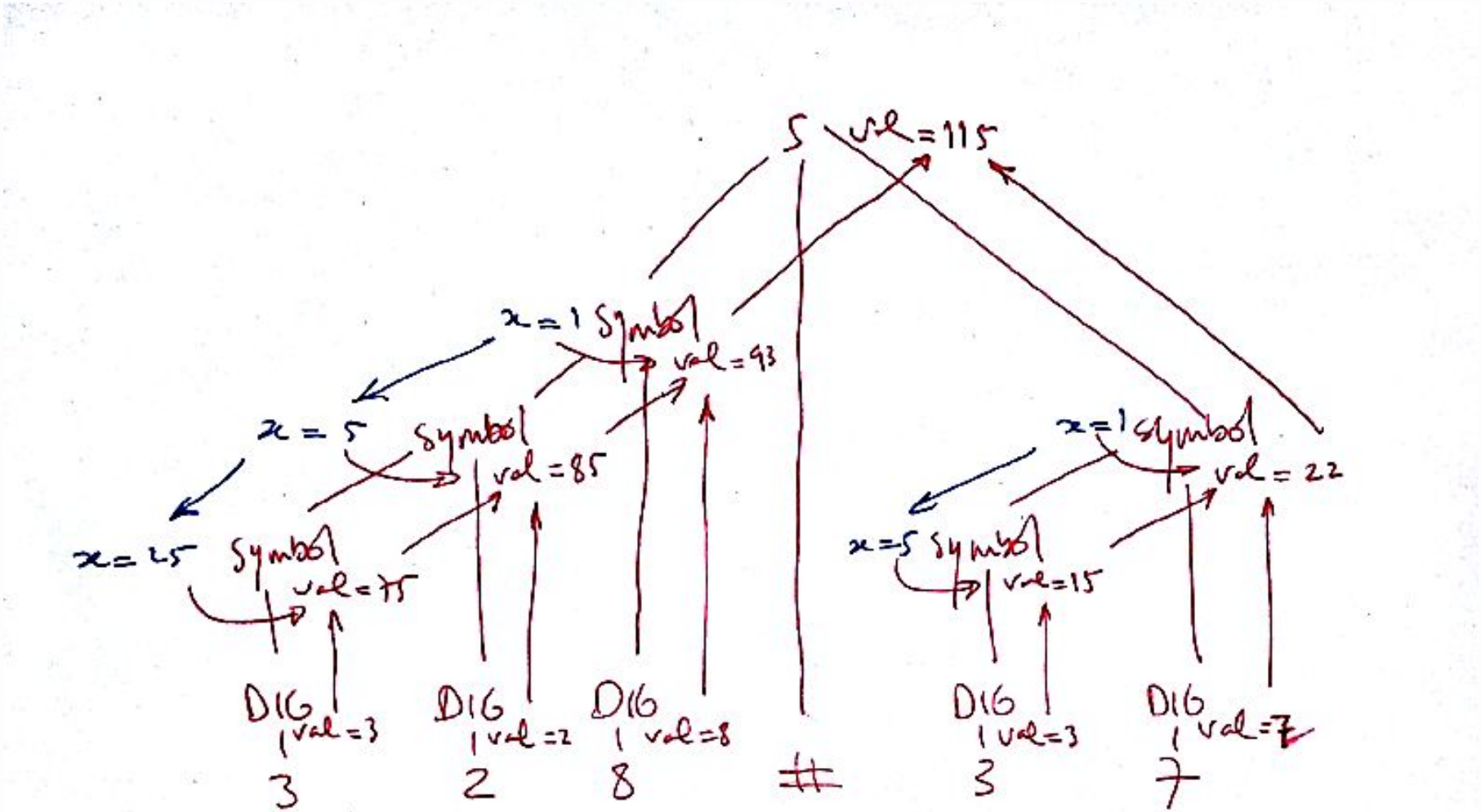
Gramática de L-atributos:

- Os valores dos atributos dependem apenas:
  - dos valores dos atributos à sua esquerda na produção
  - dos valores herdados da cabeça da regra
- Os valores dos atributos podem ser avaliados da esquerda para a direita (*left-to-right*)

Pode ser usado na análise sintáctica descendente (Parsing LL).

Grafo de dependências

Exemplo – Gramática de L-atributos

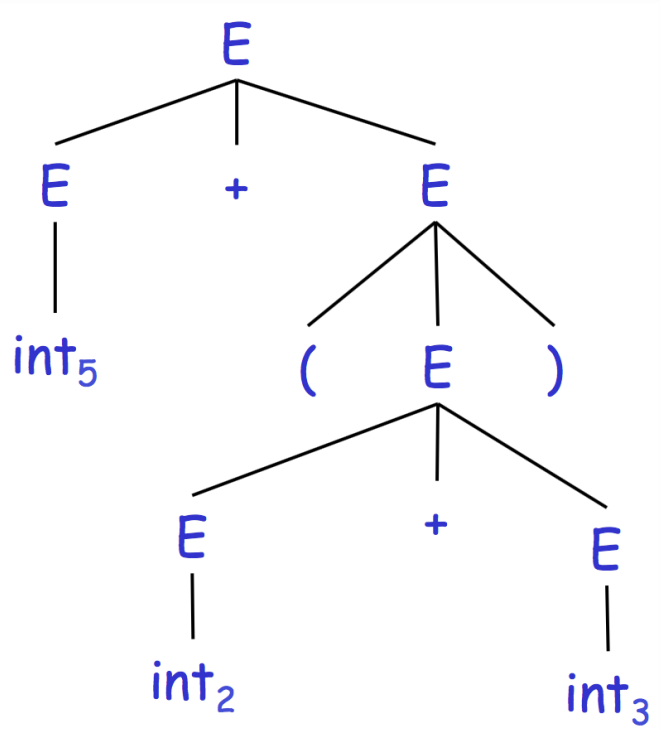


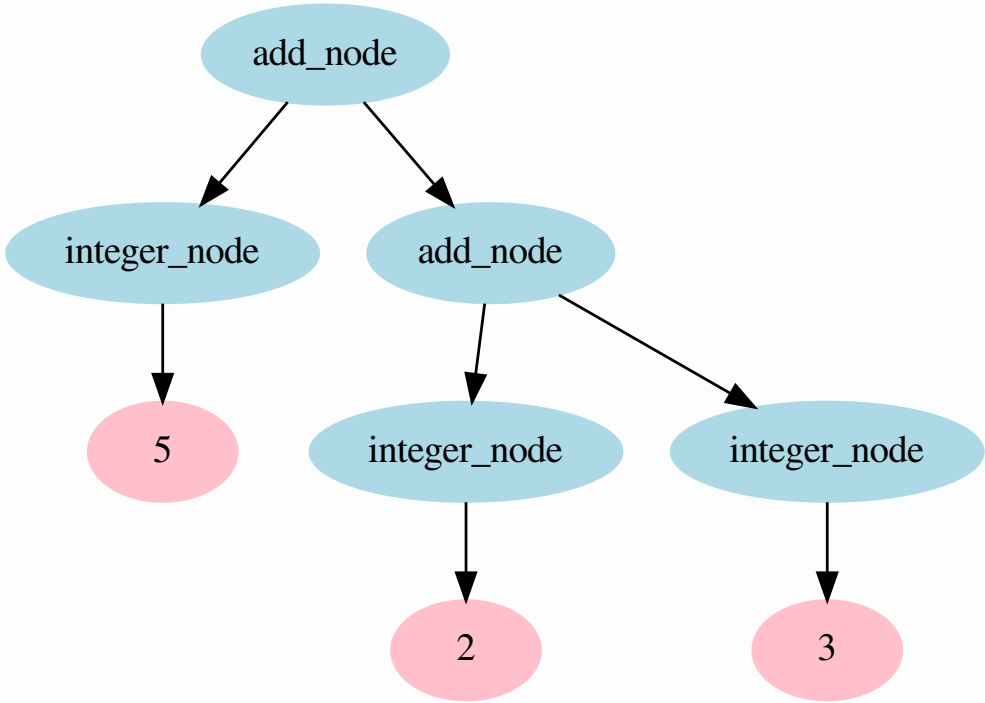
Outline

Abstract Syntax Tree - AST

- Representação da estrutura do programa
  - Difere da árvore de parsing!
- Serve para análise semântica
  - Utilização de 1 ou mais passos sobre a AST
- Serve para geração de código intermédio

Árvores de parsing vs. ASTs



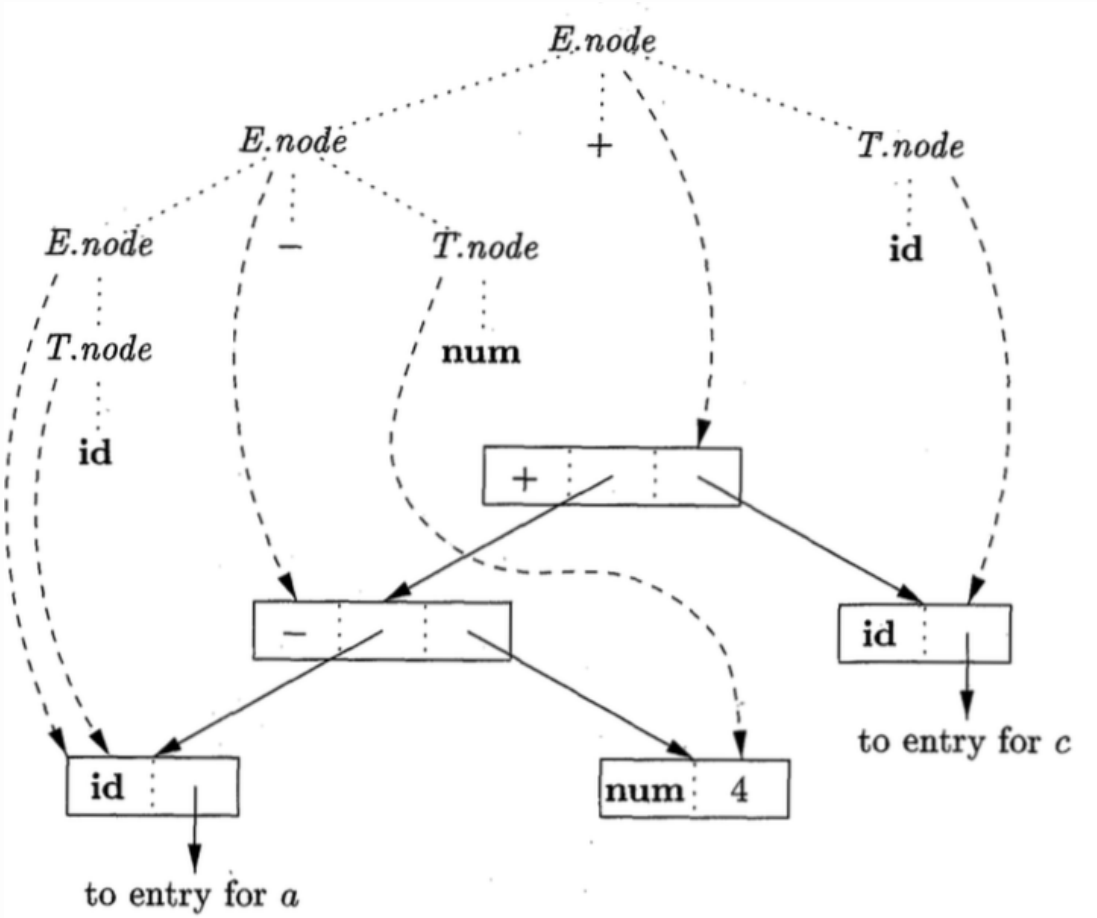


- ASTs eliminam detalhe desnecessário
- Estrutura de dados essencial em compiladores

Aplicações de SDTs – ASTs

	PRODUCTION	SEMANTIC RULES
1)	$E \rightarrow E_1 + T$	$E.node = \textbf{new}\ Node('+', E_1.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = \textbf{new}\ Node('-', E_1.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
6)	$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Aplicações de SDTs – ASTs



1) $p_1 = \textbf{new}\ \mathit{Leaf}(\textbf{id}, \mathit{entry\text{-}a});$
2) $p_2 = \textbf{new}\ \mathit{Leaf}(\textbf{num}, 4);$
3) $p_3 = \textbf{new}\ \mathit{Node}('-', p_1, p_2);$
4) $p_4 = \textbf{new}\ \mathit{Leaf}(\textbf{id}, \mathit{entry\text{-}c});$
5) $p_5 = \textbf{new}\ \mathit{Node}('+', p_3, p_4);$

Outline

Exercício – 2º Teste 2010/2011

Problema

Pretende-se criar uma gramática atributiva que calcule no símbolo inicial o valor das expressões fornecidas. As expressões são codificadas: (ii) como somas ou subtracções unárias de uma unidade a uma expressão (respectivamente, $>$ e $<$); (iii) somas ou subtracções binárias (respectivamente, # e !) de expressões; ou (iii) como percentagens de expressões (indicada como uma expressão, após o operador @). Os operadores unários têm precedência sobre todos os operadores binários. O operador @ tem precedência superior aos operadores # e !. Todos os operadores binários são associativos à esquerda.

Exemplos:

- 37 é representado pela sequência DIG DIG (o primeiro token tem o atributo *val* com valor 3 e o segundo com o valor 7).
- 4$>$ tem o valor 5
- $<$3 tem o valor 2
- 36@20 tem o valor 7.2
- (24$>$#25)@20#7!$<$15#6 tem o valor 9

https://web.tecnico.ulisboa.pt/~david.matos/w/pt/index.php/Attribute_Grammars/Exercise_8:_Arithmetic

Exercício – 2º Teste 2010/2011

Gramática atributiva

Questão: Identifique a gramática atributiva correspondente ao problema.

Exercício – 2º Teste 2010/2011

Gramática atributiva

Questão: Identifique a gramática atributiva correspondente ao problema.

$P \rightarrow A$ { std::cout << "RESULT: " << (P.val = A.val) << std::endl; }

$A_0 \rightarrow A_1 \# M$ { $A_0.val = A_1.val + M.val$; }
$A_0 \rightarrow A_1 \,!\, M$ { $A_0.val = A_1.val - M.val$; }
$A \rightarrow M$ { $A_0.val = M.val$; }

$M_0 \rightarrow M_1 @ I$ { $M_0.val = M_1.val * I.val/100$; }
$M \rightarrow I$ { $M_0.val = I.val$; }

$I \rightarrow T >$ { $I.val = T.val + 1$; }
$I \rightarrow < T$ { $I.val = T.val - 1$; }
$I \rightarrow T$ { $I.val = T.val$; }

$T \rightarrow N$ { $T.val = N.val$; }
$T \rightarrow (A)$ { $T.val = A.val$; }

$N \rightarrow DIG$ { $N.val = DIG.val$; }
$N_0 \rightarrow N_1 DIG$ { $N_0.val = N_1.val * 10 + DIG.val$; }

Exercício – 2º Teste 2010/2011

Gramática atributiva

Questão: Identifique a gramática atributiva correspondente ao problema.

$P \rightarrow A$	{ std::cout << "RESULT: " << (P.val = A.val) << std::endl; }
$A_0 \rightarrow A_1 \# M$	$\{ A_0.val = A_1.val + M.val; \}$
$A_0 \rightarrow A_1 \,!\, M$	$\{ A_0.val = A_1.val - M.val; \}$
$A \rightarrow M$	$\{ A_0.val = M.val; \}$
$M_0 \rightarrow M_1 @ I$	$\{ M_0.val = M_1.val * I.val/100; \}$
$M \rightarrow I$	$\{ M_0.val = I.val; \}$
$I \rightarrow T >$	$\{ I.val = T.val + 1; \}$
$I \rightarrow < T$	$\{ I.val = T.val - 1; \}$
$I \rightarrow T$	$\{ I.val = T.val; \}$
$T \rightarrow N$	$\{ T.val = N.val; \}$
$T \rightarrow (A)$	$\{ T.val = A.val; \}$
$N \rightarrow DIG$	$\{ N.val = DIG.val; \}$
$N_0 \rightarrow N_1 DIG$	$\{ N_0.val = N_1.val * 10 + DIG.val; \}$

Que tipo de gramática obteve?

Exercício – 2º Teste 2010/2011

Gramática atributiva

Questão: Identifique a gramática atributiva correspondente ao problema.

$P \rightarrow A$	{ std::cout << "RESULT: " << (P.val = A.val) << std::endl; }
$A_0 \rightarrow A_1 \# M$	$\{ A_0.val = A_1.val + M.val; \}$
$A_0 \rightarrow A_1 \,!\, M$	$\{ A_0.val = A_1.val - M.val; \}$
$A \rightarrow M$	$\{ A_0.val = M.val; \}$
$M_0 \rightarrow M_1 @ I$	$\{ M_0.val = M_1.val * I.val/100; \}$
$M \rightarrow I$	$\{ M_0.val = I.val; \}$
$I \rightarrow T >$	$\{ I.val = T.val + 1; \}$
$I \rightarrow < T$	$\{ I.val = T.val - 1; \}$
$I \rightarrow T$	$\{ I.val = T.val; \}$
$T \rightarrow N$	$\{ T.val = N.val; \}$
$T \rightarrow (A)$	$\{ T.val = A.val; \}$
$N \rightarrow DIG$	$\{ N.val = DIG.val; \}$
$N_0 \rightarrow N_1 DIG$	$\{ N_0.val = N_1.val * 10 + DIG.val; \}$

Que tipo de gramática obteve? Gramática S-atributos

Exercício – 2º Teste 2010/2011

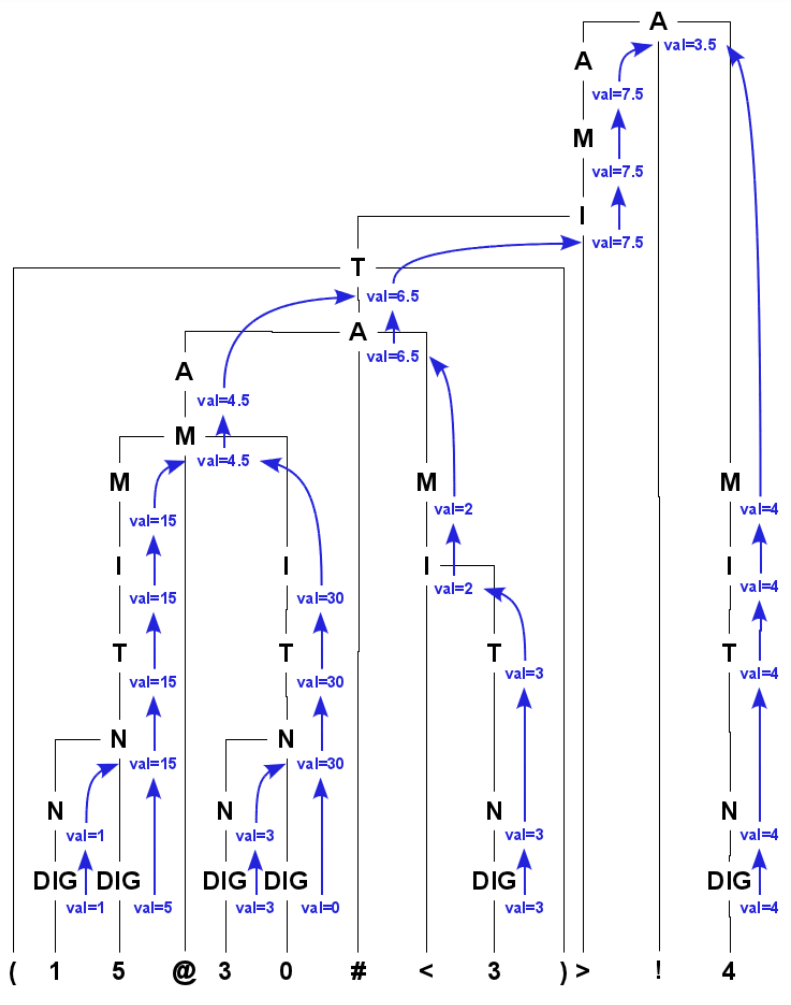
Árvore sintática decorada e grafo de dependências

Entrada: `(15@30#<3)>!`

Exercício – 2º Teste 2010/2011

Árvore sintática decorada e grafo de dependências

Entrada: `(15@30#<3)>!`



Exercício – 2º Teste 2010/2011

Especificação Flex

```
%option debug 8bit noyywrap
%{
   #include "y.tab.h"
%}

%%

[[:digit:]]  yylval.i = atoi(yytext); return DIG;
[@()<>!#]    return *yytext;
.|\n         ;

%%
```

Exercício – 2º Teste 2010/2011

Especificação BISON

```
%{
   #include <cstdlib>
   #include <iostream>
    inline void yyerror(const char *msg) { std::cout << msg << std::endl; }
%}
%union { int i; double d; }
%token<i> DIG
%type<d> expr num
%left '#' '!'
%left '@'
%nonassoc '<' '>'
%%
print: expr { std::cout << "RESULT: " << $1 << std::endl;}
     ;

expr: num            { $$ = $1; }
    | '<' expr       { $$ = $2 - 1; }
    | expr '>'       { $$ = $1 + 1; }
    | expr '#' expr  { $$ = $1 + $3; }
    | expr '!' expr  { $$ = $1 - $3; }
    | expr '@' expr  { $$ = $1 * $3 / 100; }
    | '(' expr ')'   { $$ = $2; }
    ;

num : DIG      { $$ = $1; }
    | num DIG  { $$ = 10 * $1 + $2; }
    ;
%%
extern int yylex();
extern int yyparse();
int main() { return yyparse(); }
```

Exercício – Expressões com bases

Problema

Pretende-se criar uma gramática atributiva que some uma sequência de inteiros separados por +. A expressão tem no início a base de trabalho (valor entre 2 e 36 indicado em base 10: sequência de dígitos entre 0 e 9, representados pelo elemento lexical DIG), sendo separada da soma propriamente dita pelo símbolo #. Os números a somar (entre parenteses) são codificados como sequências de dígitos de 0 a 9 e, para bases acima de 10, contendo também as letras de A a Z que forem necessárias para representar todos os valores da base de trabalho.

Exemplo: `1 7 # ( 1 2 4 + 1 G )`

- Crie a gramática para realizar a função descrita.
- Construa a árvore semântica anotada para a entrada acima (incluindo grafo de dependências).

https://web.tecnico.ulisboa.pt/~david.matos/w/pt/index.php/Attribute_Grammars/Exercise_5:_Expressions_with_bases_(2)

Exercício – Expressões com bases

Gramática atributiva

Exercício – Expressões com bases

Árvore sintática decorada e grafo de dependências

Entrada: `1 7 # ( 1 2 4 + 1 G )`

Exercícios

Mais exemplos

https://web.tecnico.ulisboa.pt/~david.matos/w/pt/index.php/Attribute_Grammars

Outline

Analisador sintáctico Bison/Yacc

Introdução

- Yacc - Yet another compiler-compiler
  - Stephen Johnson em 1975
  - Copyright Bell Labs/AT&T
- Reimplementações populares:
  - Berkeley yacc (byacc) (Robert Corbett, 1989)
    - Re-implementação com algoritmo de construção mais eficiente
    - Licença dominio público
  - Bison (Robert Corbett, 1985)
    - Extensão compatível com YACC (Richard Stallman).
    - Licença GNU GPL-3

Analisador sintáctico Bison/Yacc

Bison:
Dada a especificação de uma gramática, gera código capaz de organizar os tokens da entrada numa árvore sintáctica de acordo com a gramática.
Bison é compatible com Yacc.

- Gramática especificada em Backus-Naur Form (BNF)
  - cada regra está associada a uma acção semântica
  - As acções semânticas são executadas quando cada nó é **reduzido** (i.e., quando todo o corpo foi visto)
- Parser gerado é do tipo LALR(1) (Look-Ahead LR)
  - Para além de LALR(1), e a diferença do Yacc, o Bison é capaz de gerar outros parsers (ex: canonical LR(1)).

Analisador sintáctico Bison/Yacc

Estrutura da especificação

```
%{
código de preparação
%}
definições
%%
regras e acções semânticas
%%
código
```

Três secções separadas por uma linha, apenas com os caracteres %%

- Código de preparação é adicionado ao topo do `file.tab.c`
- Definições e Regras vão definir a função `yyparse()` do `file.tab.c`
- Código é adicionado ao fim do `file.tab.c`

Analisador sintáctico Bison/Yacc

Estrutura - Código de preparação

O código de preparação pode conter:

- includes (e.g. `#include <iostream>`)
- declaração de variáveis globais
- definição de funções auxiliares
- macros
- ...

Analisador sintáctico Bison/Yacc

Estrutura - Definições

Definições podem incluir:

- Definição de símbolos terminais (usados Flex & YACC)
  `%token tWHILE tIF tPRINT tREAD tBEGIN tEND`

Analisador sintáctico Bison/Yacc

Estrutura - Definições

Definições podem incluir:

- Definição de símbolos terminais (usados Flex & YACC)
  `%token tWHILE tIF tPRINT tREAD tBEGIN tEND`
- Tipos disponíveis para os símbolos (terminais ou não terminais)
  `%union { ... };` (slide seguinte)

Analisador sintáctico Bison/Yacc

Estrutura - Definições

Definições podem incluir:

- Definição de símbolos terminais (usados Flex & YACC)
  `%token tWHILE tIF tPRINT tREAD tBEGIN tEND`
- Tipos disponíveis para os símbolos (terminais ou não terminais)
  `%union { ... };` (slide seguinte)
- Tipificação de símbolos terminais
  `%token<s> tIDENTIFIER tSTRING`

Definições podem incluir:

- Definição de símbolos terminais (usados Flex & YACC)
  `%token tWHILE tIF tPRINT tREAD tBEGIN tEND`
- Tipos disponíveis para os símbolos (terminais ou não terminais)
  `%union { ... };` (slide seguinte)
- Tipificação de símbolos terminais
  `%token<s> tIDENTIFIER tSTRING`
- Tipificação de símbolos não terminais
  `%type<lvalue> lval`

Analisador sintáctico Bison/Yacc

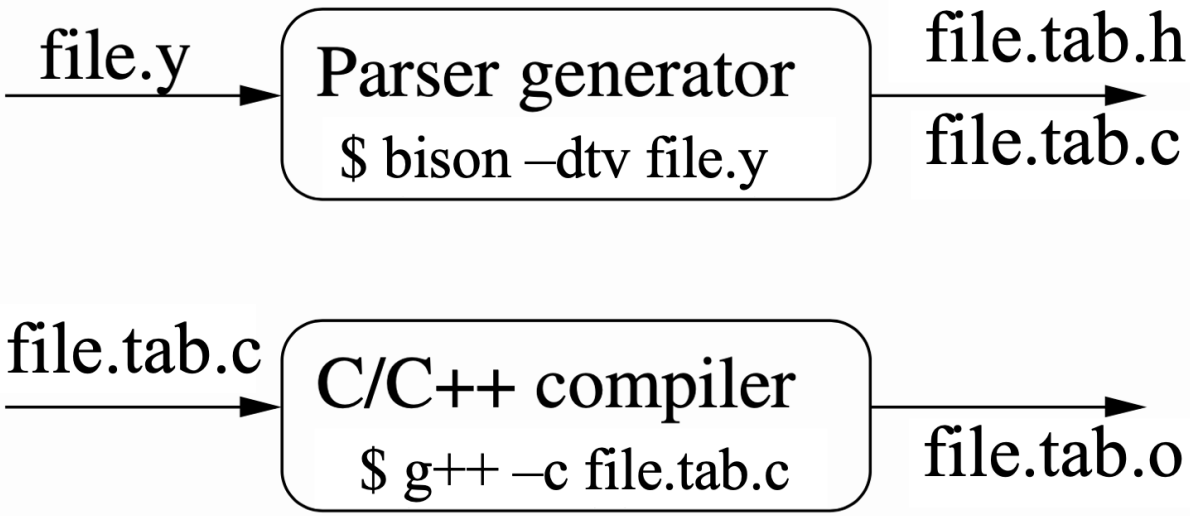
Estrutura - %union

Tipos disponíveis para os símbolos (terminais ou não terminais)

```
%union {
  int                 i;          /* XPL example */
  std::string        *s;          /* symbol name */
  cdk::basic_node    *node;       /* node pointer */
  cdk::sequence_node *sequence;   /* sequence node */
  cdk::expression_node *expression; /* expression node */
  cdk::lvalue_node   *lvalue;     /* lvalue node */
};
```

- União de todos os tipos dado que cada *token* corresponde apenas a um dos casos
- Cada novo tipo de *token* (símbolo terminal) ou nó da árvore (símbolo não terminal), tem de ser declarado na `%union`

Analisador sintáctico Bison/Yacc



- ficheiro `file.tab.c` com o parser gerado
- **-t** inclusão de instruções de debug no código compilado
- **-v** ficheiro `file.output` com descrição do parser gerado
- **-d** ficheiro `file.tab.h` com identificação de tokens e os tipos

Analisador sintáctico Bison/Yacc

simple_parser.tab.h

```
/* Token kinds.  */
#ifndef YYTOKENTYPE
# define YYTOKENTYPE
  enum yytokentype
  {
    YYEMPTY = -2,
    YYEOF = 0,                     /* "end of file"  */
    YYerror = 256,                 /* error  */
    YYUNDEF = 257,                 /* "invalid token"  */
    tINTEGER = 258,                /* tINTEGER  */
    tIDENTIFIER = 259,             /* tIDENTIFIER  */
    tSTRING = 260,                 /* tSTRING  */
    tWHILE = 261,                  /* tWHILE  */
    tIF = 262,                     /* tIF  */
    tPRINT = 263,                  /* tPRINT  */
    tREAD = 264,                   /* tREAD  */
    tBEGIN = 265,                  /* tBEGIN  */
    tEND = 266,                    /* tEND  */
    tIFX = 267,                    /* tIFX  */
    tELSE = 268,                   /* tELSE  */
    tGE = 269,                     /* tGE  */
    tLE = 270,                     /* tLE  */
    tEQ = 271,                     /* tEQ  */
    tNE = 272,                     /* tNE  */
    tUNARY = 273                   /* tUNARY  */
  };
  typedef enum yytokentype yytoken_kind_t;
#endif
```

Analisador sintáctico Bison/Yacc

simple_parser.tab.h

```
union YYSTYPE
{
#line 17 "simple_parser.y"

  //--- don't change *any* of these: if you do, you'll break the compiler.
  YYSTYPE() : type(cdk::primitive_type::create(0, cdk::TYPE_VOID)) {}
  ~YYSTYPE() {}
  YYSTYPE(const YYSTYPE &other) { *this = other; }
  YYSTYPE& operator=(const YYSTYPE &other) { type = other.type; return *this; }

  std::shared_ptr<cdk::basic_type> type;        /* expression type */
  //--- don't change *any* of these --- END!

  int                   i;      /* integer value */
  std::string          *s;      /* symbol name or string literal */
  cdk::basic_node      *node;   /* node pointer */
  cdk::sequence_node   *sequence;
  cdk::expression_node *expression; /* expression nodes */
  cdk::lvalue_node     *lvalue;

#line 100 "simple_parser.tab.h"

};
typedef union YYSTYPE YYSTYPE;
```

Analisador sintáctico Bison/Yacc

Interligação com o Flex

```
%{
#include "simple_parser.tab.h"
%}
%x X_STRING
%%

"--".*                  ; /* ignore comments */
">="                    return tGE;
"<="                    return tLE;
"=="                    return tEQ;
"!="                    return tNE;
"while"                 return tWHILE;
"if"                    return tIF;
"else"                  return tELSE;
"print"                 return tPRINT;
"read"                  return tREAD;
"begin"                 return tBEGIN;
"end"                   return tEND;
[A-Za-z][A-Za-z0-9_]*   yylval.s = new std::string(yytext); return tIDENTIFIER;
\'                      yy_push_state(X_STRING); yylval.s = new std::string("");
<X_STRING>\'            yy_pop_state(); return tSTRING;
<X_STRING>\\\'          *yylval.s += yytext + 1;
<X_STRING>.             *yylval.s += yytext;
<X_STRING>\n            yyerror("newline in string");
[0-9]+                  yylval.i = strtol(yytext, nullptr, 10); return tINTEGER;
[-()<>=+*/%;{}.]        return *yytext;
[ \t\n]+                ; /* ignore whitespace */
.                       yyerror("Unknown character");

%%
```

Uma especificação Bison/Yacc tem associado pares:

Regra { Acção semântica }

- o corpo da Regra é constituído por zero ou mais símbolos terminais e não terminais
  - $A \rightarrow \epsilon$ é representado pelo corpo vazio
- a Acção (código C/C++ arbitrário)

Uma especificação Bison/Yacc tem associado pares:

Regra { Acção semântica }

- o corpo da Regra é constituído por zero ou mais símbolos terminais e não terminais
  - $A \rightarrow \epsilon$ é representado pelo corpo vazio
- a Acção (código C/C++ arbitrário)

A pilha do Bison:

- Contém todos os símbolos do corpo
- Quando a avaliação do corpo chega ao fim
  - é feito pop de todos os símbolos do corpo
  - é feito push do símbolo da cabeça da regra
  - é avaliado o atributo da cabeça da regra

Comunicação entre as acções e o parser, é feita através do símbolo "$"

- $1, $2, ..., $n refere-se ao 1º, 2º, ..., nº símbolo do corpo
- $$ refere-se ao valor do símbolo não terminal na cabeça da regra
- Por omissão, se a acção semântica for vazia, o valor atribuído ao símbolo na cabeça da regra é o valor do 1º símbolo do corpo ($$ = $1)

Analisador sintáctico Bison/Yacc

Estrutura - Regras/Acções

```
list : stmt          { $$ = new cdk::sequence_node(LINE, $1); }
     | list stmt     { $$ = new cdk::sequence_node(LINE, $2, $1); }
     ;
stmt : expr ';'                              { $$ = new simple::evaluation_node(LINE, $1); }
     | tPRINT expr ';'                       { $$ = new simple::print_node(LINE, $2); }
     | tREAD lval ';'                        { $$ = new simple::read_node(LINE, $2); }
     | tWHILE '(' expr ')' stmt              { $$ = new simple::while_node(LINE, $3, $5); }
     | tIF '(' expr ')' stmt %prec tIFX      { $$ = new simple::if_node(LINE, $3, $5); }
     | tIF '(' expr ')' stmt tELSE stmt      { $$ = new simple::if_else_node(LINE, $3, $5, $7); }
     | '{' list '}'                          { $$ = $2; }
     ;
expr : tINTEGER                { $$ = new cdk::integer_node(LINE, $1); }
     | tSTRING                 { $$ = new cdk::string_node(LINE, $1); }
     | '-' expr %prec tUNARY   { $$ = new cdk::neg_node(LINE, $2); }
     | expr '+' expr           { $$ = new cdk::add_node(LINE, $1, $3); }
     | expr '-' expr           { $$ = new cdk::sub_node(LINE, $1, $3); }
     | expr '*' expr           { $$ = new cdk::mul_node(LINE, $1, $3); }
     | expr '/' expr           { $$ = new cdk::div_node(LINE, $1, $3); }
     | expr '%' expr           { $$ = new cdk::mod_node(LINE, $1, $3); }
     | expr '<' expr           { $$ = new cdk::lt_node(LINE, $1, $3); }
     | expr '>' expr           { $$ = new cdk::gt_node(LINE, $1, $3); }
     | expr tGE expr           { $$ = new cdk::ge_node(LINE, $1, $3); }
     | expr tLE expr           { $$ = new cdk::le_node(LINE, $1, $3); }
     | expr tNE expr           { $$ = new cdk::ne_node(LINE, $1, $3); }
     | expr tEQ expr           { $$ = new cdk::eq_node(LINE, $1, $3); }
     | '(' expr ')'            { $$ = $2; }
     | lval                    { $$ = new cdk::rvalue_node(LINE, $1); }
     | lval '=' expr           { $$ = new cdk::assignment_node(LINE, $1, $3); }
     ;
lval : tIDENTIFIER             { $$ = new cdk::variable_node(LINE, $1); }
     ;
```

Precedências/associatividades:

- Associados a *tokens* na secção Declarações
- Usados na resolução de conflictos/ambiguidades

Especificado com linhas iniciadas com `%left`, `%right` ou `%nonassoc`

- Todos os *tokens* na mesma linha têm o mesmo nível de precedência/associatividade
- Linhas subsequentes têm maior precedência/associatividade
- `%left` - define *tokens* associativos à esquerda
- `%right` - define *tokens* associativos à direita
- `%nonassoc` - define *tokens* que não se podem associar com eles próprios

Exemplo:

```
%nonassoc tIFX
%nonassoc tELSE

%right '='
%left tGE tLE tEQ tNE '>' '<'
%left '+' '-'
%left '*' '/' '%'
%nonassoc tUNARY
```

Input: $a = b = c * d - e - f * g$

lido como: $a = (b = (((c * d) - e) - (f * g)))$

Analisador sintáctico Bison/Yacc

Estrutura - Precedências

`%prec` - muda o nível de precedência associado a uma regra

- Aparece imediatamente depois do corpo da regra
- Seguido de um *token*
- Faz com que a regra fique com a mesma precedência do token

Analisador sintáctico Bison/Yacc

Estrutura - Precedências

`%prec` - muda o nível de precedência associado a uma regra

- Aparece imediatamente depois do corpo da regra
- Seguido de um *token*
- Faz com que a regra fique com a mesma precedência do token

```
%right '='
%left tGE tLE tEQ tNE '>' '<'
%left '+' '-'
%left '*' '/' '%'
%nonassoc tUNARY
%%

expr : tINTEGER                 { $$ = new cdk::integer_node(LINE, $1); }
     | tSTRING                  { $$ = new cdk::string_node(LINE, $1);  }
     | '-' expr %prec tUNARY    { $$ = new cdk::neg_node(LINE, $2);     }
     | expr '+' expr            { $$ = new cdk::add_node(LINE, $1, $3); }
     | expr '-' expr            { $$ = new cdk::sub_node(LINE, $1, $3); }
     | expr '*' expr            { $$ = new cdk::mul_node(LINE, $1, $3); }
 ...
```

- O despiste de problemas em especificações Flex e Bison pode ser realizado acrescentando ao ficheiro de especificação Flex (`.l`):
  - A opção debug no início: `%option debug`
  - A seguente ação antes da primeira regra:

```
...
%%
                                {yydebug=1;}
expr : tINTEGER                 { $$ = new cdk::integer_node(LINE, $1); }
...
```

- O desenvolvimento da gramática nos compiladores simples abordados deve ser realizado de forma incremental:
  - Maior facilidade de deteção de possíveis conflitos.
  - A opção `-Wcounterexamples` do Bison permite gerar exemplos dos conflitos.

https://web.tecnico.ulisboa.pt/~david.matos/w/pt/index.php/The_YACC_Parser_Generator/Example:_Calculator_with_Variables

Uma calculadora simples tem um número não especificado de variáveis inteiras e os operadores inteiros binários comuns (ou seja, adição, subtração, multiplicação, divisão e módulo) e operadores inteiros unários (`+` e `-`).
A linguagem contêm os seguintes conceitos (tokens): `VAR` (uma variável: o atributo `s` correspondente conterá seu nome); `INT` (um inteiro: o atributo `i` correspondente contém seu valor); e os operadores.
Múltiplas operações são separadas por `,` ou `:`, neste caso, mostrando o resultado pela saída estândar.

Analisador sintáctico Bison/Yacc

Mais exemplos

https://web.tecnico.ulisboa.pt/~david.matos/w/pt/index.php/The_YACC_Parser_Generator

Questões?

Dúvidas?