COMPILADORES 2019/2020

aula 0x02 - Análise Lexical e Sintáctica

27/02/2019

Pedro Patinho <pp@di.uevora.pt> [CLAV-256]

Universidade de Évora - Departamento de Informática

SUMÁRIO

ANÁLISE LEXICAL

ANÁLISE LEXICAL

- Primeiro passo da compilação
- Consiste em ler o código-fonte e detectar "palavras"
  - Input é uma *stream* de bytes
  - O analisador lexical é um autómato finito
  - Usam-se ferramentas para gerar o analisador (e.g., *flex*)
  - Para o reconhecimento de padrões definem-se *expressões regulares*

    `[0-9]+` (números inteiros)
    `\"[^\"]*\"` (strings)
    `[ \t]+` (espaços e tabs)
- Output é uma stream de *Tokens*

TOKENS AO PORMENOR

- Um *token* é uma "palavra"
- No caso de palavras reservadas e pontuação, contém apenas o "nome" do símbolo
- No caso de nomes ou números, contém o "tipo" de símbolo e o "valor"
- Comentários e espaços em branco são ignorados, mas interessam para localização no ficheiro fonte

TOKEN - PROGRAMATICAMENTE

```
1       struct Token {
2         enum { TOK_SYM, TOK_ID, TOK_INTLIT,
3                TOK_FLOATLIT, TOK_STR, TOK_BOOLLIT } kind;
4         union {
5           int intlit;
6           float floatlit;
7           char *id;
8           char *str;
9           char sym;
10        } u;
11      };  /* where is boollit??? */
12      . . .
13      struct Token T;
14      T.kind = TOK_SYM;
15      T.u.sym = '+';
```

EXEMPLO - CALCULADORA

```
1   %{
2   #include <stdlib.h>
3   #include "parser.h"
4
5   %}
6
7   INT [0-9]+
8   ID  [a-zA-Z][a-zA-Z0-9_]*
9
10
11  %%
12
13  {INT}     { yylval.val = atof(yytext); return NUM; }
14  "+"   { return ADD; }
15  "-"   { return SUB; }
16  "/"   { return DIV; }
17  "*"   { return MUL; }
18  "("   { return LPAR; }
19  ")"   { return RPAR; }
20  "="   { return ASSIGN; }
21  "quit"    { exit(0); }
22  {ID}      { yylval.id = strdup(yytext); return ID; }
23  \n    { return NL; }
24  .     {/*ignorar*/}
25
26  %%
27
28  int yywrap()  {return 1;}
```

ANÁLISE LEXICAL ISOLADA

Apesar de o analisador lexicar normalmente trabalhar em conjunto com o analisador sintáctico, há problemas possíveis de resolver apenas com o analisador lexical!

Exemplo: calculadora RPN (Reverse Polish Notation)

`3 4 + 5 *` $\rightarrow$ (35)

ANÁLISE SINTÁCTICA

ANALISADOR SINTÁCTICO (PARSER) - O QUE É?

- Autómato finito
- Usa gramática para verificar sintaxe do código fonte
  - Input = Tokens
  - Output = APT (Abstract Parse Tree)
- Gerado por uma ferramenta (e.g., *bison*)
- A ocorrência de um erro implica o fim do processo, na maioria das vezes.

ACÇÕES SEMÂNTICAS

Produção de nós

```
. . .
exp : exp ADD exp { $$ = mknode(ND_ADD, $1, $3); }
    | exp MUL exp { $$ = mknode(ND_MUL, $1, $3); }
. . .
```

- Acções activadas aquando da escolha de uma produção (regra gramatical)
- Local adequado para executar o "trabalho" do compilador
  - Execução imediata (só para interpretadores)
  - Produção de estruturas de dados (nós da APT)

PRODUÇÃO DE NÓS - SINTAXE ABSTRACTA

- requer "tipagem" dos símbolos da gramática
  - tipo *grande* (classe...) para regras
  - tipo *pequeno* (construtor...) para produções
- assume construção "bottom-up"

TIPOS DE NÓS

```
. . .
%union {
    double val;
    char *name;
    calc_t_exp exp;
    calc_t_seq seq;
}

%token <val>NUM
%token <name>ID
. . .
%type <exp>exp
%type <seq>seq
```

APT (ABSTRACT PARSE TREE)

ABSTRACT PARSE TREE

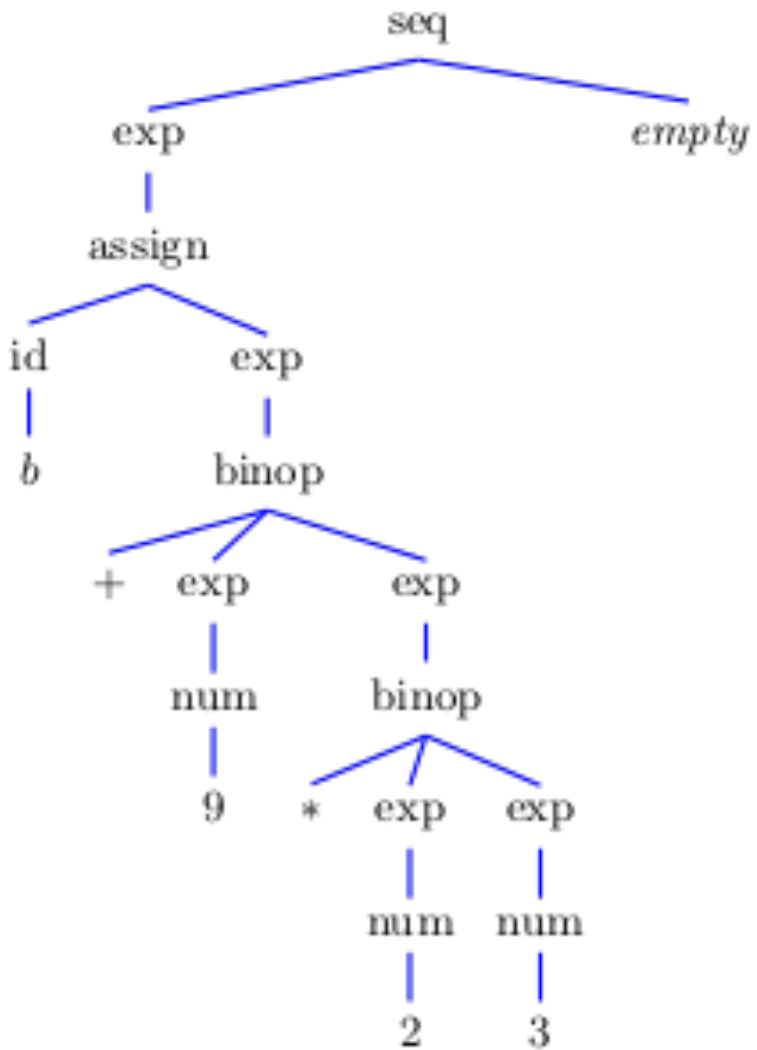
- Representa o input, mas em forma de árvore
- Estrutura próxima da sintaxe concreta, mas mais leve
- Muito ligada às regras e produções da gramática
- Descrição pode ser ambígua
  - Não é usada para fazer parse
  - Usa o parse já efectuado

SINTAXE ABSTRACTA

- Simplificação das “classes” do input
- Várias opções de representação

APT - EXEMPLO

```
b = 9 + 2 * 3
```



APT - CONVENÇÕES DE CONSTRUÇÃO

- Depende sempre da gramática
- Escolher uma representação
  - Eficaz
  - Completa
  - Conveniente

- Estruturas a representar:
  - Literais
  - Identificadores (variáveis, funções)
  - Expressões
  - Tipos
  - Instruções (statements, declarações)

EXEMPLO

APT - OUTRO EXEMPLO

```
120 + 3 * 2 + 1; b = 9 + 2 + 3; c = (b + 1) * b
```