

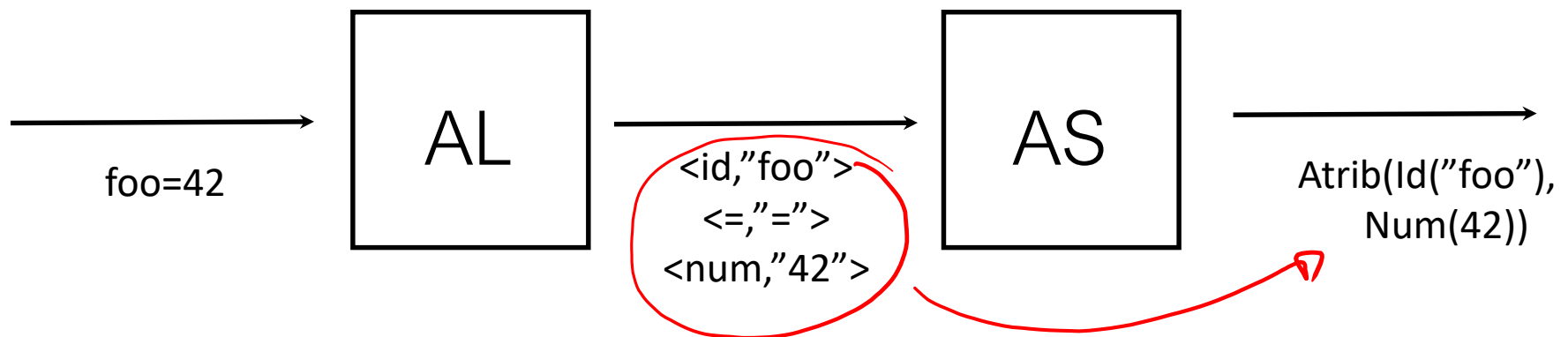
Compiladores - Especificando Sintaxe

Fabio Mascarenhas – 2018.1

<http://www.dcc.ufrj.br/~fabiom/comp>

Análise Sintática

- A análise sintática agrupa os tokens em uma *árvore sintática* de acordo com a estrutura do programa (e a gramática da linguagem)
- Entrada: sequência de tokens fornecida pelo analisador léxico
- Saída: árvore sintática do programa



Gramáticas como especificação

- Usamos regras envolvendo expressões regulares e tokens para especificar o analisador léxico de uma linguagem de programação
- Para especificar o analisador sintático, vamos usar regras envolvendo gramáticas livres de contexto
- Na gramática de uma linguagem, os tipos de tokens são os terminais, e os não-terminais dão as estruturas sintáticas da linguagem: comandos, expressões, definições...

Padrões gramaticais

- É muito comum a sintaxe de uma linguagem de programação ter *listas*, ou sequências, de alguma estrutura sintática
- Expressamos essas listas na gramática com recursão à esquerda ou recursão à direita:

IDS \rightarrow IDS , id

IDS \rightarrow id

IDS \rightarrow id , IDS

IDS \rightarrow id

- A escolha de recursão à esquerda ou direita vai dar a forma da árvore resultante, mas em uma árvore abstrata normalmente usamos uma lista diretamente

Listas

- Para o caso geral, se E é a estrutura sintática que estamos repetindo, e SEP é o separador da lista, uma lista de E s é:

$$\left\{ \begin{array}{l} ES \rightarrow ES \text{ SEP } E \\ ES \rightarrow E \end{array} \right. \quad \left| \quad \begin{array}{l} ES \rightarrow E \text{ SEP } ES \\ ES \rightarrow E \end{array}$$

- Notem que a lista não pode ser vazia; caso queiramos uma lista vazia precisamos de um outro não-terminal que pode ser ou vazio ou ES
- Repetição é tão comum em gramáticas que existe uma notação para isso: $\{ t \}$ é uma sequência de 0 ou mais ocorrências do termo t . Agora podemos expressar uma lista potencialmente vazia diretamente:

$$\begin{array}{l} ES \rightarrow E \{ SEP E \} \\ ES \rightarrow \end{array}$$

Opcional

- Um outro padrão recorrente na sintaxe são termos opcionais, como o bloco else de um comando if. Podemos expressá-los com uma regra vazia, ou com duas versões de cada regra que contém o termo opcional:

IF -> if EXP then BLOCO ELSE end

ELSE -> else BLOCO

ELSE ->

IF -> if EXP then BLOCO else BLOCO end

IF -> if EXP then BLOCO end

- Novamente, existe uma notação especial [t] para um termo opcional:

IF -> if EXP then BLOCO [else BLOCO] end

EBNF, alternativa e agrupamento

- Os meta-símbolos {} e [] fazem parte da notação EBNF para gramáticas, uma forma mais fácil de escrever gramáticas para linguagens de programação
- Outras facilidades da EBNF são o uso de | para indicar várias possibilidades sem precisar de múltiplas regras, e () para agrupamento
- Naturalmente quando usamos EBNF precisamos de alguma forma de separar os meta-símbolos do seu uso como tokens da linguagem! Podemos por os tokens entre aspas simples, por exemplo:

CMD -> print EXP | id = EXP

EXP -> T { + T | - T }

T -> id | num | '(' EXP ')'

CMD \rightarrow print EXP

CMD \rightarrow id = EXP

EXP \rightarrow EXP + T
EXP \rightarrow EXP - T
EXP \rightarrow T

T \rightarrow id

T \rightarrow num

T \rightarrow (EXP)

TINY

- Uma linguagem simples usada no livro texto alternativo:

S -> CMDS

CMDS -> CMD { ; CMD }

CMD -> if EXP then CMDS [else CMDS] end

| repeat CMDS until EXP

| id := EXP

| read id

| write EXP

EXP -> SEXP [< SEXP | = SEXP]

SEXP -> TERMO { + TERMO | - TERMO }

TERMO -> FATOR { * FATOR | / FATOR }

FATOR -> "(" EXP ")" | num | id

TINY

- Uma linguagem simples usada no livro texto alternativo:

S -> CMDS

CMDS -> CMD { ; CMD }

CMD -> if COND then CMDS [else CMDS] end

| repeat CMDS until COND

| id := EXP

| read id

| write EXP

COND -> EXP (< EXP | = EXP)

EXP -> TERMO { + TERMO | - TERMO }

TERMO -> FATOR { * FATOR | / FATOR }

FATOR -> "(" EXP ")" | num | id