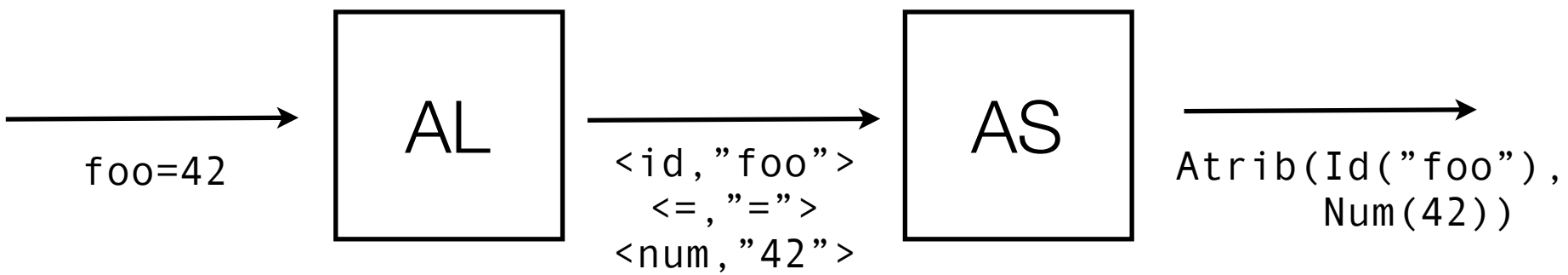
Compiladores - Especificando Sintaxe

Fabio Mascarenhas - 2013.1

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 -> IDS , id
IDS -> id

IDS -> id , IDS
IDS -> 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:

```
ES -> ES SEP E
ES -> E
```

- 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:

```
ES -> E { SEP E }
ES ->
```

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 possiblidades 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 ‘)’
```

TINY

- Uma linguagem simples usada no livro texto:

```
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:

```
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
```

Analisador Recursivo

- Maneira mais simples de implementar um analisador sintático a partir de uma gramática, mas não funciona com muitas gramáticas
- A ideia é manter a lista de tokens em um vetor, e o token atual é um índice nesse vetor
- Um **terminal** testa o token atual, e avança para o próximo token se o tipo for compatível, ou falha se não for
- Uma **sequência** testa cada termo da sequência, falhando caso qualquer um deles falhe
- Uma **alternativa** guarda o índice atual e testa a primeira opção, caso falhe volta para o índice guardado e testa a segunda, assim por diante

Analisador Recursivo

- Um **opcional** guarda o índice atual, e testa o seu termo, caso ele falhe volta para o índice guardado e não faz nada
- Uma **repetição** repete os seguintes passos até o seu termo falhar: guarda o índice atual e testa o seu termo
- Um **não-terminal** vira um procedimento separado, e executa o procedimento correspondente
- Construir a árvore sintática é um pouco mais complicado, as alternativas, opcionais e repetições devem jogar fora nós da parte que falhou!

Um analisador recursivo para TINY

- Vamos construir um analisador recursivo para TINY de maneira sistemática, gerando uma árvore sintática

- O vetor de tokens vai ser gerado a partir de um analisador léxico escrito com o JFlex

Retrocesso local x global

- O retrocesso em caso de falha do nosso analisador é *local*. Isso quer dizer que se eu tiver `( A | B ) C` e `A` não falha mas depois `C` falha, ele não tenta `B` depois `C` novamente
- Da mesma forma, se eu tenho `A | A B` a segunda alternativa nunca vai ser bem sucedida
- As alternativas precisam ser *exclusivas*
- Retrocesso local também faz a repetição ser *gulosa*
- Uma implementação com retrocesso *global* é possível, mas mais complicada

Detecção de erros

- Um analisador recursivo com retrocesso também tem um comportamento ruim na presença de erros sintáticos
- Ele não consegue distinguir *falhas* (um sinal de que ele tem que tentar outra possibilidade) de *erros* (o programa está sintaticamente incorreto)
- Uma heurística é manter em uma variável global uma marca d'água que indica o quão longe fomos na sequência de tokens

Recursão à esquerda

- Outra grande limitação dos analisadores recursivos é que as suas gramáticas não podem ter *recursão à esquerda*

- A presença de recursão à esquerda faz o analisador entrar em um laço infinito!

- Precisamos transformar recursão à esquerda em repetição

- Fácil quando a recursão é direta:

```
A -> A x1 | ... | A xn | y1 | ... | yn
```

```
A -> ( y1 | ... | yn ) { x1 | ... | xn }
```

Eliminação de recursão sem EBNF

```
A -> A x1
...
A -> A xn
A -> y1
...
A -> yn
```



```
A  -> y1 A'
...
A  -> yn A'
A' -> x1 A'
...
A' -> xn A'
A' ->
```