Compiladores

Linguagem X++

Baseado no livro do Prof. Delamaro – Como Construir um Compilador

Convenção

- $\langle symbol \rangle$ representa o não-terminal $symbol$;
- símbolos terminais são representados com ”symbol” ;
- metassímbolos da notação BNF são representados grafados com [ ].

$$\langle methodbody \rangle \quad \rightarrow \quad "(" \ \langle paramlist \rangle \ ")" \ \langle statement \rangle$$

$$\langle returnstat \rangle \quad \rightarrow \quad "return" \ [\ \langle expression \rangle \ ]$$

Símbolo Inicial

- `<program> → [<classlist>]`
- Esse não-terminal indica que um programa em X++ é composto de uma lista de classes ou se trata de uma cadeia vazia.
- Assim, um programa salvo em um arquivo sem nada escrito é um programa válido.

Lista de Classes

- O não-terminal `classlist` representa uma repetição de declarações de classe, que basicamente é composto de uma seqüência de declarações de classe como:

```
class a { .... }
class b { .... }
class c { .... }
```

<classlist>

- <classlist> → (<classdecl>)$^{+}$
- <classlist> → (<classdecl>)$^{+}$
  [<classlist>]

Declaração de Classe <classdecl>

- `<classdecl>` → `“class” “ident” [“extends” “ident”] <classbody>`
- Esse não-terminal `classdecl` representa a declaração de uma classe.
  - Ele indica que tal estrutura se inicia com a palavra reservada `class` que vem seguida por um identificador (outro terminal) que irá dar nome à classe sendo declarada.

Declaração de Classe <classdecl>

- <classdecl> → “class” “ident”
  [“extends” “ident”]
  <classbody>
- Esse não-terminal classdecl representa a declaração de uma classe.
  - Depois do nome da classe, pode ou não, aparecer a palavra extends e o nome da superclasse da qual a classe sendo declarada descende.

Declaração de Classe <classdecl>

```
class a ....
```

ou

```
class a extends b .....
```

Corpo da Classe <classbody>

- <classbody> → “{” [<classlist>]
  (<vardecl> “;”)*
  (<constructdecl>)*
  (<methoddecl>)* “}”
- Ele se inicia com um { que (opcionalemente) vem seguido por um não-terminal classlist, declarado anteriormente.
- Isso significa que a linguagem permite a declaração de classes aninhadas.
- Depois, seguem-se as declarações de variáveis, construtores e métodos.

Declaração de Variáveis <vardecl>

- Uma declaração de variáveis, representada pelo não terminal `vardecl`, é semelhante à declaração de variáveis em Java.
  - int a;
  - String a,b;
  - mytype a[], b[][];
- <vardecl> → (“int” | “string” | ”ident”) “ident” (“[“ | “]”)* (“,” “ident”(“[“”]”)*)*

Declaração de Variáveis <vardecl>

- Essa produção gera cadeias que começam com o tipo da variável a ser declarada, que pode ser int, string, ou um identificador que é o nome de uma classe declarada no próprio programa.
- Em seguida, vêm os nomes das variáveis sendo declaradas.
- Podem ser seguidos ou não por colchetes que indicam a dimensão de cada variável.

Declaração de construtores e métodos

- `<constructdecl>` → `“constructor” <methodbody>`
- `<methodbody>` → `(“int” | “string” | “ident”) (“[“”]”)* “ident” <methodbody>`
- A declaração de um construtor começa com a palavra `constructor`. Um `constructor` não tem tipo de retorno ou um nome.

Declaração de construtores e métodos

- <constructdecl> → “constructor” <methodbody>
- <methodbody> → (“int” | “string” | “ident”) (“[“”]”)* “ident” <methodbody>
- A declaração de um método inicia-se com o tipo de retorno do método, que pode ou não ter várias dimensões, seguido pelo nome do método e pelo seu corpo.
  - int[] fatorial ....

Corpo de método (construtor)

```
int[ ] fatorial (int x, string z) {
    comando 1;
    comando 2;
    .....
}
constructor (int x, string z) {
    comando 1;
    comando 2;
    .....
}
```

Corpo de método (ou construtor)

- <methodbody> → “(“<paramlist>”)”<statement>
- <paramlist> → [(“int”|”string”|”ident”)
  “ident” (“[“”]”)*
  (“,”(“int” | “string” |
  “ident”)
  “ident” (“[“”]”)*)*]
- int a
- int a, string b
- string b[][], int a, MyType c

Comandos da Linguagem

- Declaração de variáveis locais: `int a, b, c;`
- Comando de atribuição: `a = b + c`
- Comando de impressão: `print`
- Comando de leitura: `read`
- Comando de término do método e retorno de valor: `return`
- Comando de seleção: `if`
- Comando de repetição: `for`
- Comando de interrupção: `break`
- Comando composto: `{ ... }`
- Comando vazio: `;`

Comando <statement>

- <statement> → (
  <vardecl> “;” |
  <atribstat> “;” |
  <printstat> “;” |
  <readstat> “;” |
  <returnstat> “;” |
  <superstat> “;” |
  <ifstat> |
  <forstat> |
  “{“<statlist> “}” |
  “break” “;” |
  “;”
)

Comando de Atribuição <atribstat>

- `<atribstat>` → `<lvalue>` ”=“
  (`<expression>` | `<alocexpression>`)

- Referência a uma posição de memória (representada pelo não-terminal lvalue), seguida por um = e, depois, uma expressão ou uma referência a um novo objeto, utilizando o operador new.

a = 0
a[10] = b + c.d
a[10].b = new MyType()

Comando de Impressão <printstat>

- <printstat> →”print“ <expression>

print 123
print a
*print a[10].b * c.d[e]*

Comando de Leitura <readstat>

- <readstat> → ”read“ <lvalue>

read a
reab a.b
read a.b[c+2]

Chamada de super-construtor <superstat>

- `<superstat>` → ”super“ “(“`<arglist>`”)”

- Nos construtores pode ser utilizado também o comando `super` para chamada do construtor da superclasse

  super()
  super(a, null)

Comando de Seleção <ifstat>

- <ifstat> → ”if“ “(“<expression>”)”
  <statement> [“else” <statement>]
- if (a > b) read b;
- if (a + b == c)

```
{
  read d;
  print d;
  return 0;
}
else
  return 1;
```

Comando de Repetição <forstat>

- <forstat> → ”for“ “(“[<atribstat>]”;”
  [<expression>] “;”
  [<atribstat>]”)”<statement>
- for(;;)
- for(a=0; ;) read b[a];
- for(a=0;a<b;a=a+1)
  {
  read c[a];
  print c[a];
  }

Outros

- <statement> → (

  <vardecl> ”;” |

  <atribstat> “;” |

  <printstat> “;” |

  <readstat> “;” |

  <returnstat> “;” |

  <superstat> “;” |

  <ifstat> |

  <forstat> |

  “{“ <statlist> “}” |

  “break” “;” |

  “;”

)

Outros

- ```
  if (a > b)
      ;
  else
      read b;
  ```
- ```
  for (a = 0; a < b; a = a + 1)
  {
      read c[a];
      if(a > b>
          break;
      else
          print c[a];
  }
  ```

Quais são válidos?

- <statlist> → <statement> [<statlist>]
- ```
  int mymethod(int a)
  {
  }
  ```
- ```
  int mymethod(int a)
  {
      ;
  }
  ```
- ```
  int mymethod(int a)
     ;
  ```
  ```

lvalue <lvalue>

- <lvalue> → “ident” (“[“ <expression> “]” |
“.” “ident” [“(“ <arglist> “)”])*

• Representa uma referência a uma posição de memória e foi usado em atribstat e readstat.
• Inicia-se sempre com um identificador que é o nome de uma variável ou um método.
• Esse identificador pode vir seguido várias vezes por um índice no caso de se estar referindo a uma variável indexada; ou
• uma referência a um campo de um objeto; ou
• um nome de um método seguido por uma lista de argumentos, no caso em que se está fazendo uma chamada de um método.

lvalue: exemplos

- `read a;` contém somente a referência a uma variável simples;
- `read a[0][1]` é uma referência a uma variável indexada;
- `read a.b` é uma referência ao campo b da variável a;
- `read a.b[0][1]` é uma combinação dos dois tipos anteriores;
- `Read a.b(12).c` é uma combinação dos três tipos. Nesse caso, a é uma variável que referencia um objeto. Com esse objeto estamos invocando o método b, passando 12 como argumento. Esse método deve retornar um outro objeto do qual estamos referenciando o campo c.

lvalue impróprio

• Utilizando este não-terminal num comando de atribuição poderíamos ter a.b(12) = 10, o que é ilegal pois estaríamos tentando atribuir valor a uma chamada de método. Sintaticamente isso, porém, será permitido na nossa linguagem. Tal erro será apontado em fases posteriores da análise.b

Criação de objetos

- <alocexpression> → “new”
  (“ident” “(“ <arglist> “)” |
  (“int” | “string” | “ident”)
  (“[“ <expression> “]”)$^{+}$)

```
new MyType(10, a * b)
new string[10][i][k]
```

Expressões

- <expression> → <numexpr> [
  ["<" | ">" | "<=" | ">=" |
  "==" | "!=")
  <numexpr>]
- <numexpr> → <term> (("+" | "-") <term>)*

- <term> → <unaryexpr> (
  ("*" | "/" | "%")
  <unaryexpr>)*

Expressões

- <unaryexpr> → [(“+” | “-”)] <factor>

- <factor> → (“int-constant” | “string-constant”
  | “null” | <lvalue>
  | “(“ <expression> “)”)

- <arglist> → [<expression> (“,” <expression>)*]