ThrPy——一种多线程式的类 Python 语言

ZY2006109 - 姬轶
ZY2006321 - 刘卓程

8th December 2020

1 语言设计的背景及范型

1.1 设计背景

Python 是一门解释型语言，在神经网络方面有着丰富的库函数，神经网络具有很强的并行性和适应性，可以应用于控制、信息、预测等许多领域。基于多线程技术，神经网络的并行性理论上可以高效完成计算任务。多线程技术能够使程序的响应速度更快，并且占用大量处理时间的任务使用多线程可以提高 CPU 利用率，即占用大量处理时间的任务可以定期将处理器时间让给其它任务。

但因为 Python 中的全局解释锁（Global Interpreter Lock，GIL）的存在无法使用 threading 充分利用 CPU 资源，任何线程在运行之前必须获取这个全局锁才能执行，每当执行完 100 条字节码，全局锁才会释放，切换到其他线程执行，如果想充分发挥多核 CPU 的计算能力需要使用 multiprocessing 模块 (Windows 下使用会有诸多问题)。这些问题使得多线程的使用名存实亡。且过多种类的库使得相关学习者在初入门时上手较为困难，这些问题让 Python 的效率大大降低。

1.2 语言特性

因此，我们希望设计一种更为轻便且高效的类 Python 语言，使得多线程能够真正实现，从而增加该语言的易用性，让使用者能够更简便的开发。因此命名为 ThrPy，多线程操作的底层实现是通过调用一个 C 库函数来实现，在 C 库中再调用标准的 pthread_create 函数来实现。ThrPy 语言提供的主要特性有：

1. 完备的数据类型：基础数据类型主要参考 Python 的数据类型，并在此基础上定义了线程规格中所需的数据类型。
2. 对于线程声明有严格的结构化形式，灵活的控制结构：对每个线程都以 thread identifier 和 end identifier 来标识，其中参数需要按着结构进行定义。
3. 程序拥有更快的响应速度，由于实现了多线程的存在，充分利用了多核 CPU 的计算能力，使得程序在运行上拥有了更快的响应速度。

2 语言的语法、语义规格说明

2.1 词法规则

2.1.1 关键字

本语言额外定义了 18 个关键字：thread, features, flows, properties, end, none, in, out, data, port, event, parameter, flow , source, sink , path, constant , access。

关键字是保留字，并且必须是小写。

2.1.2 专用符号

本语言定义了 8 个专用符号：=> +=> ; : :: { } ->

2.1.3 数值类型

本语言中，主要数据类型为标示符 identifier 和浮点数 decimal，其词法规通过下列此法规则进行定义：

identifier = identifier_letter (underline?letter_or_digit)*

identifier_letter=a|..|z|A|..|Z|

letter_or_digit = identifier_letter | digit

digit = 0|..|9

underline=_

decimal = sign? numeral . numeral

numeral= digit (digit)*

sign = + | -

2.2 语法规则

本语言采用 EBNF 来描述语法，并通过对部分语法的改造，消除间接递归的文法可能导致的死循环问题。同时，为了降低语法的复杂度，允许一些语法存在二义性，这部分问题可以通过定义优先级从而消除，具体可以参照 Python 的运算符优先级和结合性一览表。线程规格语法定义如下，其中加粗部分为终结符。

多线程使用 thread identifier 和 end identifier 作为关键字定义一个线程，对于每一个线程，可以声明其特征、数据流、属性相关规格。

总体语法规则如下：

ThreadSpec → **thread identifier** [**feature** featureSpec] [**flows** flowSpec] [**properties** association;] **end identifier** ;

featureSpec → portSpec | ParameterSpec | **none** ;

portSpec → **identifier :** IOtype portType [**{** { association } **}**] ;

portType → **data port** [reference] | **event data port** [reference] | **event port**

ParameterSpec → **identifier :** IOtype **parameter** [reference] [**{** { association } **}**] ;

IOtype → **in | out | in out**

flowSpec → flowSourceSpec | flowSinkSpec | flowPathSpec | **none**;

flowSourceSpec → **identifier : flow source identifier** [**{** { association } **}**] ;

flowSinkSpec → **identifier : flow sink identifier** [**{** { association } **}**] ;

flowPathSpec → **identifier : flow path dentifier -> identifier**;

association → **identifier** [**::** **identifier**] splitter [**constant**] **access decimal** | **none**

splitter → **=>** | **+=>**

reference → [{ **identifier ::** }] **identifier**

2.2.1 线程规格定义

ThreadSpec → **thread identifier** [**feature** featureSpec] [**flows** flowSpec] [**properties** association;] **end identifier** ;

指称语义

需要使用的下层辅助函数有:

```
new_thread(environment, store) 在 environment 和 store 中开启一个新的 thread;
upadte_thread_feature(thread, featureSpec) 将 thread 的 feature 属性改为 featureSpec;
update_thread_flows(thread, flowSpec) 将 thread 的 flows 属性改为 flowSpec;
update_thread_association(thread, association) 将 thread 的 properties 属性改为
    association ;
find(environment, store, id) 在 environment, store 中找到 thread_id 的地址;
```

```
execute: ThreadSpec -> (Environment -> Store -> Store)
execute [T] env sto = sto'

execute [Th Id] env sto =
      let thr = new_thread(env, sto) in
      let variable loc = find(env, Id) in
      update(sto, loc, thr)

execute [Th Fe] env sto =
      let thr = update_thread_features(thr, Fe) in
      let loc = find(env, sto, thr) in
      update(sto, loc, thr)

execute [Th Fl] env sto =
      let thr = update_thread_features(thr, Fl) in
      let loc = find(env, sto, thr) in
      update(sto, loc, thr)

execute [Th As] env sto =
      let thr = update_thread_features(thr, As) in
      let loc = find(env, sto, thr) in
      update(sto, loc, thr)
```

2.2.2 特征规格定义

featureSpec → portSpec | ParameterSpec | **none** ;

对该语法进行改造:

featureSpec → **identifier :** IOtype (portSpec | ParameterSpec) | **none** ;

指称语义

需要使用的下层辅助函数有:

```
new_feature(environment, store) 在 environment 和 store 中开启一个新的 feature;
update_feature_iotype(feature, io) 将 feature 的 iotype 属性改为 io;
update_feature_potype(feature, po) 将 feature 的 portSpec 属性改为 po;
update_feature_patype(feature, pa) 将 feature 的 ParameterSpec 属性改为 pa;
```

```
execute [Fe Id Io po] env sto =
      let fe = new_feature(env, sto) in
      let variable loc = find(env, Id) in
      let update_feature_iotype(fe, Io) in
      let update_feature_potype(fe, po) in
      update(sto, loc, fe)

execute [Fe Id Io pa] env sto =
      let fe = new_feature(env, sto) in
      let variable loc = find(env, Id) in
      let update_feature_iotype(fe, Io) in
      let update_feature_patype(fe, pa) in
      update(sto, loc, fe)

execute [Fe Id Io] env sto =
      let fe = new_feature(env, sto) in
      let variable loc = find(env, Id) in
      let update_feature_iotype(fe, Io) in
      update(sto, loc, fe)

#upadte_thread_feature(thread, featureSpec) #将 thread 的 feature 属性改为 featureSpec
```

2.2.3 端口规格定义

portSpec → **identifier :** IOtype portType [**{** { association } **}**] ;

对该语法进行改造:

portSpec → portType [**{** { association } **}**] ;

指称语义

需要使用的下层辅助函数有:

```
update_port_portype(port, porttype) 将 port 的 portType 属性改为 porttype;
update_port_asso(port, asso) 将 port 的 association 属性改为 asso;
find_fea(env, sto) 在 env 和 sto 中找寻 feature;
```

```
execute [porttype] env sto =
        let fea = find_fea(env, sto) in
        let port = evaluate fea.port env sto in
        update_port_portype(port, porttype)

execute [porttype, asso] env sto =
        let fea = find_fea(env, sto) in
        let port = evaluate fea.port env sto in
        update(port, porttype)
        update_port_asso(port, asso)
```

2.2.4 端口类型定义

portType → **data port** [reference] | **event data port** [reference] | **event port**

指称语义

```
execute [port_type, ref] env sto =
        let fea = find_fea(env, sto) in
        let port = evaluate fea.port env sto in
        update_port_portype(port, port_type)
        if ref is not None
            port.reference = ref
```

2.2.5 参数规格定义

ParameterSpec → **identifier : ** IOtype **parameter** [reference] [**{** { association } **}**] ;

对该语法进行改造:

ParameterSpec → **parameter** [reference] [**{** { association } **}**] ;

指称语义

需要使用的下层辅助函数有:

```
update_port_pa(port, pa) 将 port 的 ParameterSpec 属性改为 pa;
```

```
execute [pa, ref, asso] env sto =
        let fea = find_fea(env, sto) in
        let port = evaluate fea.port env sto in
        update_port_pa(port, pa)
        if ref is not None
            let port.reference = evaluate ref env sto in
        if asso is not None
            let port.asso = evaluate asso env sto in
```

2.2.6 输入输出类型定义

IOtype → **in** | **out** | **in out**

指称语义

```
execute [io] env sto =
      let fea = find_fea(env, sto) in
      let port = evaluate fea.port env sto in
      let port.iotype = evaluate io env sto in
```

2.2.7 数据流规格定义

flowSpec → flowSourceSpec | flowSinkSpec | flowPathSpec | **none**;

对该语法进行改造:

flowSpec → **identifier : flow** (flowSourceSpec | flowSinkSpec | flowPathSpec) | **none**;

指称语义

```
new_flow(environment, store) 在 environment 和 store 中开启一个新的 flow;
update_flow_sourcespec(flow, sourcespec) 将 flow 的 flowSourceSpec 属性改为 sourcespec;
update_flow_sinkspec(flow, sinkspec) 将 flow 的 flowSinkSpec 属性改为 sinkspec;
update_flow_pathspec(flow, pathspec) 将 flow 的 flowPathSpec 属性改为 pathspec;
```

```
execute [Id flow_source_spec] env sto =
      let fl = new_flow(env, sto) in
      let update_flow_sourcespec(fl, flow_source_spec)
      let variable loc = find(env, Id) in
      update(sto, loc, fl)

execute [Id flow_sink_spec] env sto =
      let fl = new_flow(env, sto) in
      let update_flow_sinkspec(fl, flow_sink_spec)
      let variable loc = find(env, Id) in
      update(sto, loc, fl)

execute [Id flow_path_spec] env sto =
      let fl = new_flow(env, sto) in
      let update_flow_pathspec(fl, flow_path_spec)
      let variable loc = find(env, Id) in
      update(sto, loc, fl)
```

2.2.8 数据流源规格定义

flowSourceSpec → **identifier : flow source identifier** [**{** { association } **}**] ;

对该语法进行改造:

flowSourceSpec → **source identifier** [**{** { association } **}**] ;

指称语义

```
update_flow_sourcespec(flow, sourceidentifier) , 将 flow 的 flowSourceSpec 属性改为
    sourcespec;
update_flow_asso(flow, sourceidentifier, asso), 将 flow.sourceidentifier 的 association
    属性改为 asso;
find_fl(env, sto), 在 env 和 sto 中找寻 fl;
```

```
execute [sourceidentifier] env sto =
      let fl = find_fl(env, sto) in
      update_flow_sourcespec(fl, sourceidentifier)

execute [sourceidentifier, asso] env sto =
      let fl = find_fl(env, sto) in
      update_flow_sourcespec(fl, sourceidentifier)
      update_flow_asso(fl, flowSourceSpec, asso)
```

2.2.9 数据流目标规格定义

flowSinkSpec → **identifier : flow sink identifier** [**{** { association } **}**] ;

对该语法进行改造:

flowSinkSpec → sink **identifier** [**{** { association } **}**] ;

指称语义

```
update_flow_sinkspec(flow, sinkspec) , 将 flow 的 flowSinkSpec 属性改为 sinkspec;
update_flow_asso(flow, sinkspec, asso), 将 flow.sinkspec 的 association 属性改为 asso;
find_fl(env, sto), 在 env 和 sto 中找寻 fl;
```

```
execute [sinkidentifier] env sto =
      let fl = find_fl(env, sto) in
      update_flow_sinkspec(fl, sinkidentifier)

execute [sinkidentifier, asso] env sto =
      let fl = find_fl(env, sto) in
      update_flow_sourcespec(fl, sinkidentifierr)
      update_flow_asso(fl, flowSinkSpec, asso)
```

2.2.10 数据流路径规格定义

flowPathSpec → **identifier : flow path dentifier -> identifier**;

对该语法进行改造:

flowPathSpec → **path identifier -> identifier**;

指称语义

```
update_flow_path(flow, path), 修改 flow 的 path;
find_fl(env, sto), 在 env 和 sto 中找寻 fl;
```

```
execute [id1, id2] env sto =
      let fl = find_fl(env, sto) in
      let path = evaluate id1 -> id2 env sto in
      update_flow_path(f1, path)
```

2.2.11 约束定义

association → [**identifier ::**] **identifier** splitter [**constant**] **access decimal** | **none**

对该语法进行改造:

association → **identifier** [**:: identifier**] splitter [**constant**] **access decimal** | **none**

指称语义

需要使用的下层辅助函数有:

```
new_association(environment, store), 在environment和store中开启一个新的association;
reach(Id1, Id2), 判断Id2是否存在, 若存在, 在Id1中寻找Id2, 若不存在, 使用Id1;
bind_rule(id, splitter, decimal), 将identifier和其限制规则、大小绑定;
update_association(sto, association, bindrule), 将association中的规则设为bindrule;
```

```
execute [asso, id, splitter, dec] env sto =
      let asso = new_association(env, sto) in
      let variable id = reach(id1, id2)
         bind_rule(id, splitter, dec) = br in
      update_association(sto, asso, br)
```

2.2.12 分离符定义

splitter → => | +=>

指称语义

需要使用的下层辅助函数有:

```
set_limit(identifier, limit), 对identifier设置限制;
set_lower_limit(identifier, lower_limit), 对identifier设置限制下界;
update_association(sto, loc, identifier), 更新association中identifier的限制属性;
```

```
evaluate 〚id => decimal〛 =
      set_limit(id, decimal)
evaluate 〚id +=> decimal〛 =
      set_lower_limit(id, decimal)
```

2.2.13 引用定义

reference → [{ **identifier ::** }] **identifier**

对该语法进行改造:

reference → **identifier** [{ **::** **identifier** }]

指称语义

需要使用的下层辅助函数有:

```
find(source, goal), 在source中查找goal元素;
check_type(type, value), 检查 value 是否为 type 类型, 若是, 返回 value, 否则编译报错;
update_reference(reference, id), 将reference中的identifier属性设置为id;
```

```
update (stble, sto, loc) = sto [loc→stored stble]
bind_reference : Reference→(Argument→Environment)

execute[I1, I2]env sto =
       let check_type(I1, I2)
          val = evaluate I2 env sto in
       let variable loc = find(I1, I2) in
       update_reference(reference, I2)
```

3 词法、语法分析器的实现

3.1 词法分析的实现

词法分析的设计经历从正规表达式到不确定性有穷自动机（NFA）再通过词法规则的改写建立确定性有穷自动机（DFA），最后产生词法分析程序。

词法分析器对输入的语言源码进行词法分析，生成 Token 序列，并按原顺序、结构输出到 tokenOut 文件中，等待下一步的语法分析。

定义以下参数进行辅助词法分析并保存相应分析结果。

```
private static TokenType currentToken;
private static State state = State.START;
private static String rowInfo = new String();
// 错误列表
private static List<String> errorList = new ArrayList<>();
// 行Token列表
private static List<String> rowList = new ArrayList<>();
```

START：开始状态；

DONE：终止状态；

```
    while (curPosition < infoLen) {
        char c = info.charAt(curPosition);
        switch (state) {
        case START:
            start = curPosition;
            if (isIdentifierLetter(c)) {
                state = State.IL;
                currentToken = TokenType.IDENTIFIER;
            } else if (isDigit(c)) {
                state = State.DN;
                currentToken = TokenType.DECIMAL;
            } else {
                switch (c) {
                case '=':
                    currentToken = TokenType.EQUAL;
                    state = State.EN;
                    break;
                case '+':
                    currentToken = TokenType.PLUS;
                    state = State.PN;
                    break;
                case ';':
                    currentToken = TokenType.SEMI;
                    state = State.DONE;
                    break;
                case ':':
                    currentToken = TokenType.COLON;
```

```
                state = State.CN;
                break;
            case '{':
                currentToken = TokenType.LBRACE;
                state = State.DONE;
                break;
            case '}':
                currentToken = TokenType.RBRACE;
                state = State.DONE;
                break;
            case '-':
                currentToken = TokenType.MINUS;
                state = State.MN;
                break;
            case ' ':
                currentToken = TokenType.SPACE;
                state = State.START;
                break;
            case '\t':
                currentToken = TokenType.TABLE;
                state = State.START;
                break;
            case '\n':
                currentToken = TokenType.ENTER;
                state = State.START;
                break;
            default:
                state = State.DONE;
                currentToken = TokenType.ERROR;
            }
        }
        break;
    }
}
```

3.1.1 Identifier 的有穷自动机

在识别 Identifier 的过程中，设定两个状态来表示自动机的状态迁移。

IL（Identifier_Letter Next）：通过识别 Identifier_Letter 进入的状态；

UN（Underline Next）：通过识别 Underline 进入的状态；

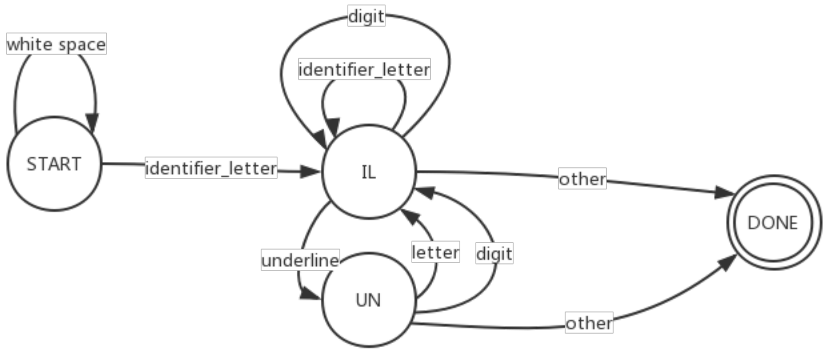

图 1: Identifier 的有穷自动机

```
case IL:
    if (isIdentifierLetter(c) || isDigit(c)) {
        state = State.IL;
        currentToken = TokenType.IDENTIFIER;
    } else if (isUnderline(c)) {
        state = State.UN;
        currentToken = TokenType.IDENTIFIER;
    } else
        state = State.DONE;
    break;
case UN:
    if (isIdentifierLetter(c) || isDigit(c)) {
        state = State.IL;
        currentToken = TokenType.IDENTIFIER;
    } else {
        state = State.DONE;
        currentToken = TokenType.ERROR;
    }
    break;
```

3.1.2 Decimal 的有穷自动机

MN（Minus Next）：通过识别–进入的状态；

PN（Plus Next）：通过识别 + 进入的状态；

DN（Digit Next）：第一次识别 Digit 循环进入的状态；

DP（Digit Point）：通过识别. 进入的状态；

DS（Digit Second）：第二次识别 Digit 循环进入的状态；

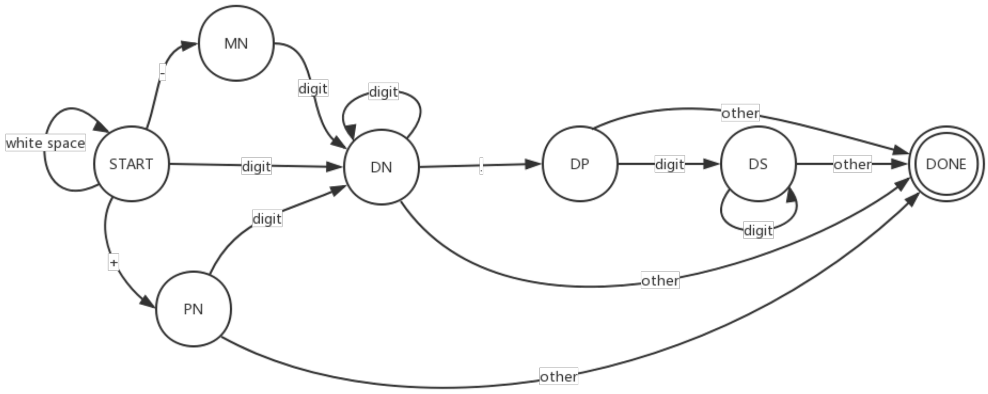

图 2: Decimal 的有穷自动机

```
case MN:
    if (isDigit(c)) {
        state = State.DN;
        currentToken = TokenType.DECIMAL;
    } else if (c == '>') {
        state = State.DONE;
        currentToken = TokenType.MINUSTO;
    } else {
        state = State.DONE;
        currentToken = TokenType.ERROR;
        curPosition--;
    }
    break;
case PN:
    if (isDigit(c)) {
        state = State.DN;
        currentToken = TokenType.DECIMAL;
    } else if (c == '=') {
        state = State.EN;
        currentToken = TokenType.PLUSEQUALTO;
    } else {
        state = State.DONE;
        currentToken = TokenType.ERROR;
        curPosition--;
    }
    break;
case DN:
    if (isDigit(c)) {
        state = State.DN;
```

```
                currentToken = TokenType.DECIMAL;
            } else if (c == '.') {
                state = State.DP;
                currentToken = TokenType.DECIMAL;
            } else {
                state = State.DONE;
                currentToken = TokenType.ERROR;
            }
            break;
        case DP:
            if (isDigit(c)) {
                state = State.DS;
                currentToken = TokenType.DECIMAL;
            } else {
                state = State.DONE;
                currentToken = TokenType.ERROR;
            }
            break;
        case DS:
            if (isDigit(c)) {
                state = State.DS;
                currentToken = TokenType.DECIMAL;
            } else
                state = State.DONE;
            break;
```

3.1.3 专有字符的有穷自动机

MN（Minus Next）：通过识别–进入的状态；

PN（Plus Next）：通过识别 + 进入的状态；

EN（Equal Next）：通过识别 = 进入的状态；

CN（Colon Next）：通过识别: 进入的状态。

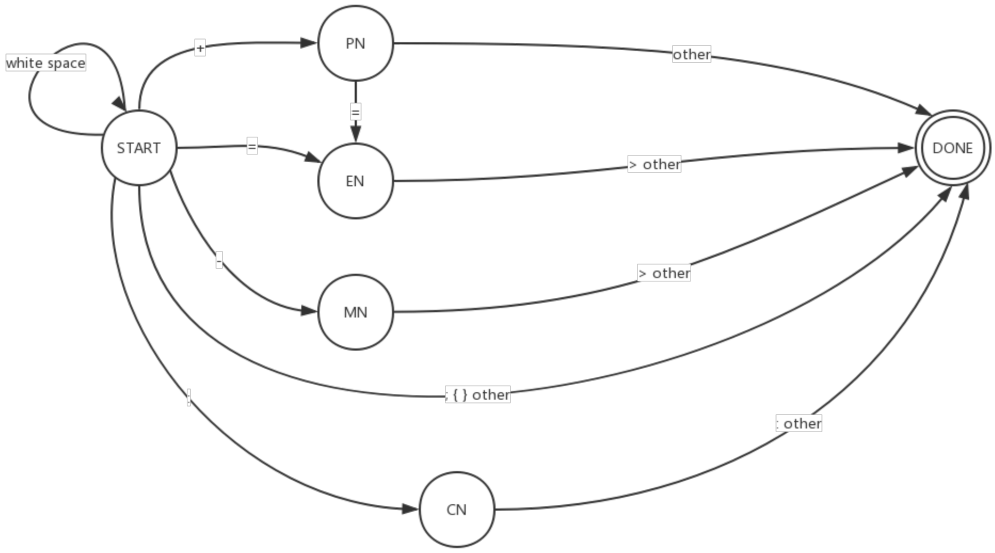

图 3: 专有字符的有穷自动机

```
case MN:
    if (isDigit(c)) {
        state = State.DN;
        currentToken = TokenType.DECIMAL;
    } else if (c == '>') {
        state = State.DONE;
        currentToken = TokenType.MINUSTO;
    } else {
        state = State.DONE;
        currentToken = TokenType.ERROR;
        curPosition--;
    }
    break;
case PN:
    if (isDigit(c)) {
        state = State.DN;
        currentToken = TokenType.DECIMAL;
    } else if (c == '=') {
        state = State.EN;
        currentToken = TokenType.PLUSEQUALTO;
    } else {
        state = State.DONE;
        currentToken = TokenType.ERROR;
        curPosition--;
    }
```

```
        break;
    case EN:
        if (currentToken == TokenType.PLUSEQUALTO && c == '>') {
            state = State.DONE;
            currentToken = TokenType.PLUSEQUALTO;
        } else if(currentToken == TokenType.EQUAL && c == '>') {
            state = State.DONE;
            currentToken = TokenType.EQUALTO;
        }else {
            state = State.DONE;
            currentToken = TokenType.ERROR;
            curPosition--;
        }
        break;
    case CN:
        if (c == ':') {
            state = State.DONE;
            currentToken = TokenType.DOUBLECOLON;
        } else {
            state = State.DONE;
            currentToken = TokenType.COLON;
        }
        break;
```

3.2 语法分析的实现

3.2.1 实现步骤

本语言实现语法分析的基本步骤：

1. 对程序设计语言的语法规则进行形式化描述（用 2 型文法）；
2. 根据语言的语法描述形式，定义各种基本语法结构的抽象语法树；
3. 选择一种合适的语法分析算法，并在分析程序中插入构造语法树等动作——语法分析程序。

3.2.2 构建抽象语法树

ThreadSpec → **thread identifier** [**feature** featureSpec] [**flows** flowSpec] [**properties** association;] **end identifier** ;

ThreadSpec（线程规格）对应抽象语法树如图 4所示：

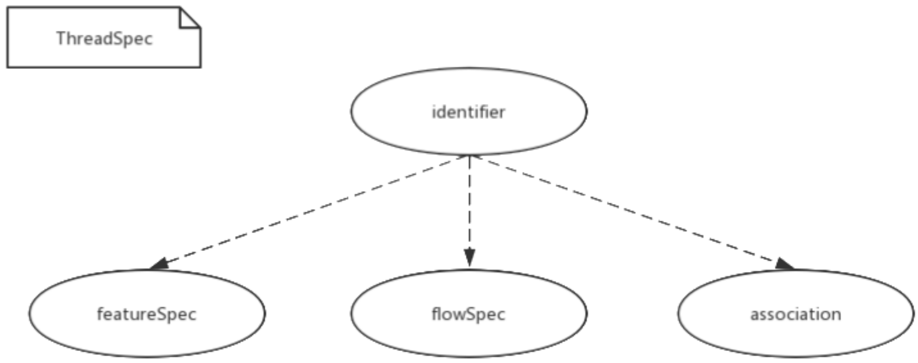

图 4: ThreadSpec（线程规格）的抽象语法树

featureSpec → **identifier :** IOtype (portSpec | ParameterSpec) | **none** ;
port → portSpec | ParameterSpec
FeatureSpec（特征规格）对应抽象语法树如图 5所示：

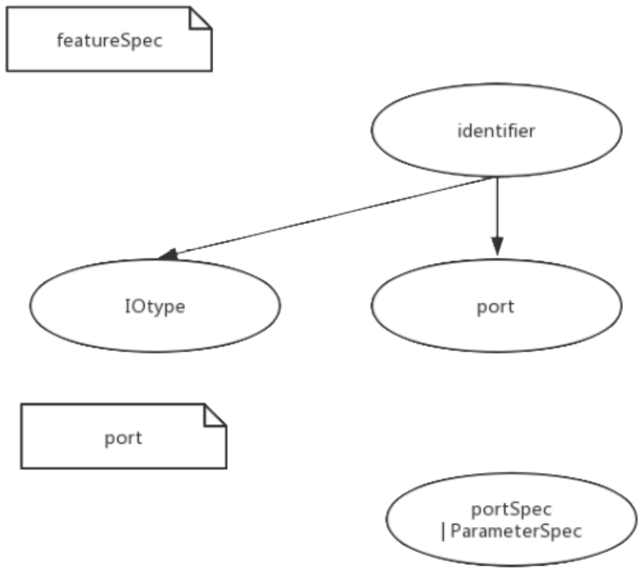

图 5: FeatureSpec（特征规格）的抽象语法树

portSpec → portType [**{** { association } **}**] ;
portSpec（端口规格）对应抽象语法树如图 6所示：

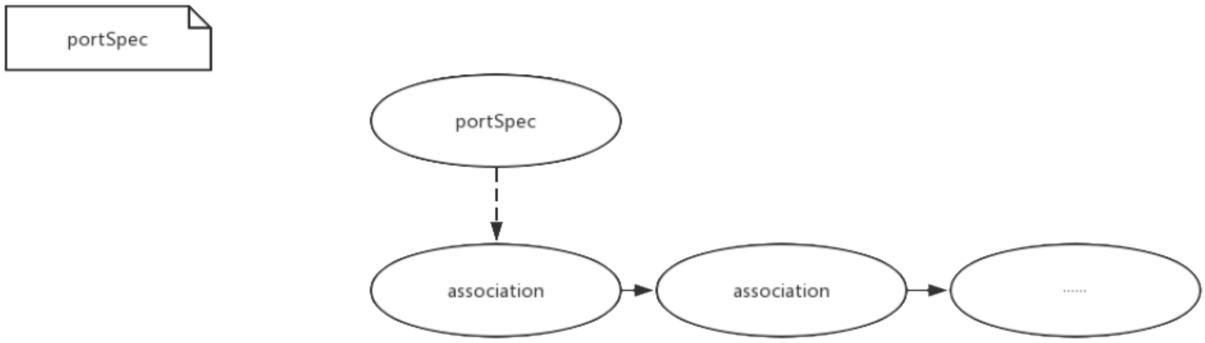

图 6: PortSpec（端口规格）的抽象语法树

portType → **data port** [reference] | **event data port** [reference] | **event port**

portType（端口类型）对应抽象语法树如图 7所示：

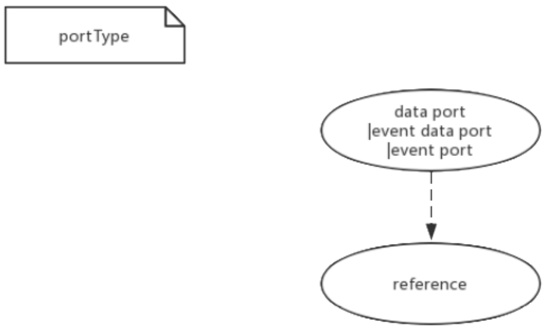

图 7: portType（端口类型）的抽象语法树

ParameterSpec → **parameter** [reference] [**{** { association } **}**] ;

ParameterSpec（参数规格）对应抽象语法树如图 8所示：

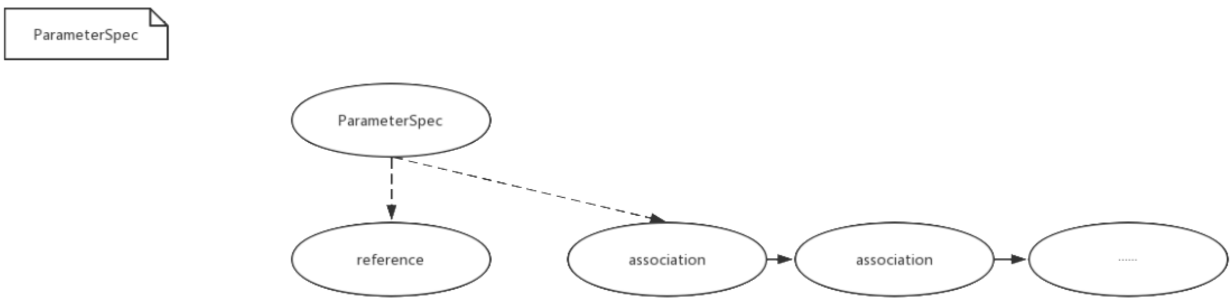

图 8: ParameterSpec（参数规格）的抽象语法树

IOtype → **in** | **out** | **in out**

IOtype（输入输出类型）对应抽象语法树如图 9所示：

图 9: IOtype（输入输出类型）的抽象语法树

flowSpec → **identifier : flow** (flowSourceSpec | flowSinkSpec | flowPathSpec) | **none**;
flowType → flowSourceSpec | flowSinkSpec | flowPathSpec

FlowSpec（数据流规格）对应抽象语法树如图 10所示：

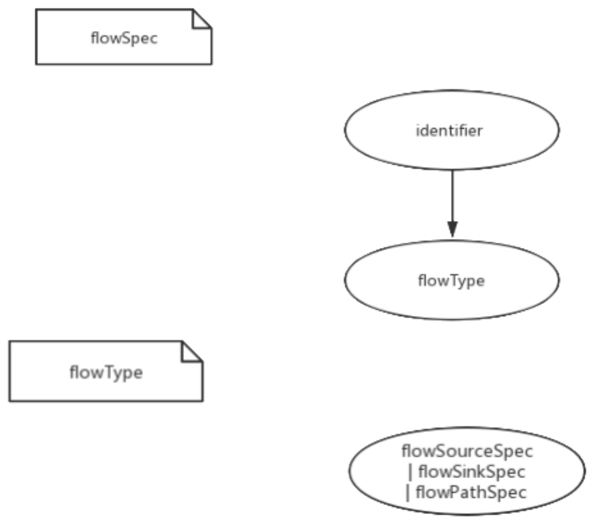

图 10: FlowSpec（数据流规格）的抽象语法树

flowSourceSpec → **source identifier** [**{** { association } **}**] ;

FlowSourceSpec（数据流源规格）对应抽象语法树如图 11所示：

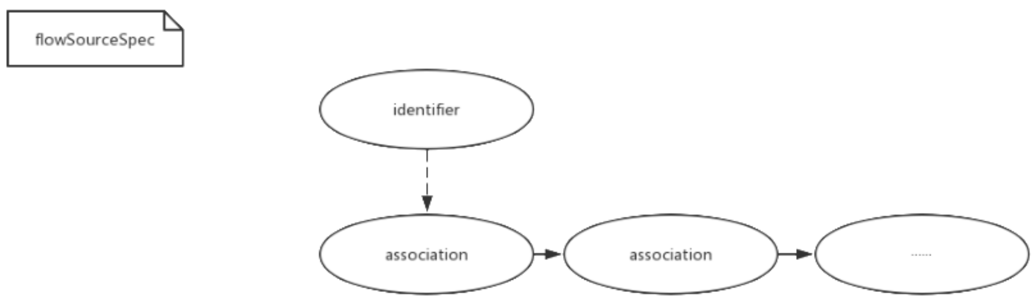

图 11: FlowSourceSpec（数据流源规格）的抽象语法树

flowSinkSpec → sink **identifier** [**{** { association } **}**] ;

FlowSinkSpec（数据流目标规格）对应抽象语法树如图 12所示：

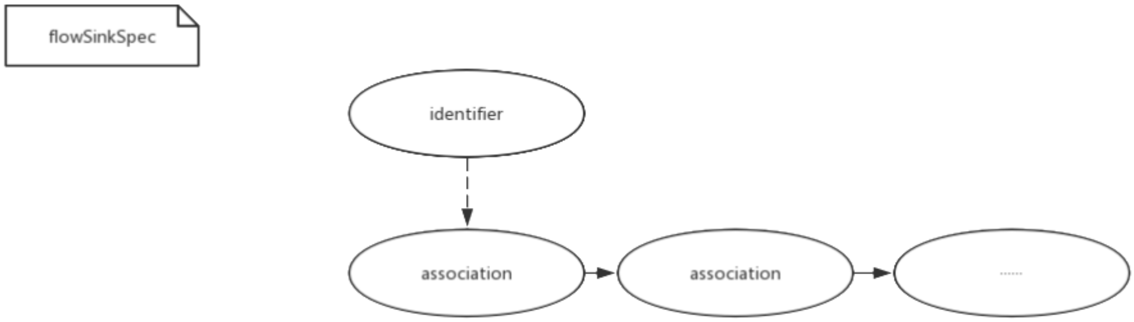

图 12: FlowSinkSpec（数据流目标规格）的抽象语法树

flowPathSpec → **path identifier -> identifier**;

FlowPathSpec（数据路径流规格）对应抽象语法树如图 13所示：

图 13: FlowPathSpec（数据路径流规格）的抽象语法树

association → **identifier** [**:: identifier**] splitter [**constant**] **access decimal** | **none**

Association（约束定义）对应抽象语法树如图 14所示：

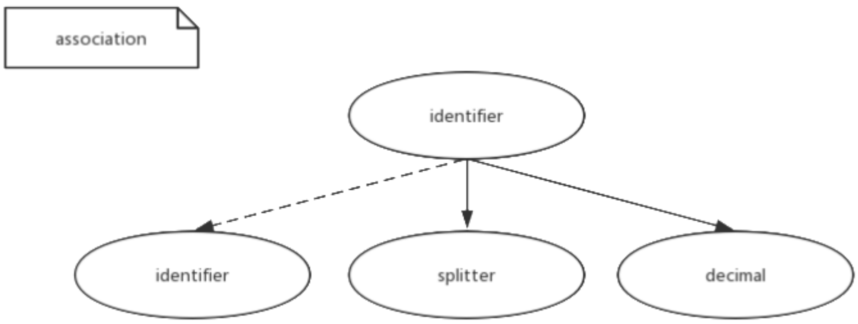

图 14: Association（数据路径流规格）的抽象语法树

splitter → => | +=>

Splitter（分隔符定义）对应抽象语法树如图 15所示：

图 15: Splitter（数据路径流规格）的抽象语法树

reference → **identifier** [{ **::** **identifier** }]

Reference（引用定义）对应抽象语法树如图 16所示：

图 16: Reference（引用定义）的抽象语法树

4 验证与测试

4.1 语言的代码范例

```
thread Thread1
features
   Position_Input : in event data port Types::Position;
flows
   flow1: flow path signal -> result1;
properties
   position_protocol => access 50.0;
end Thread1;

thread Thread_2
features
   Position_Input : in data port Types1::Types2::Position;
flows
   flow1: flow source signal {result1::result2 +=> constant access 50.0};
properties
   none;
end Thread_2;

thread Thread3d
features
   Position_Input : out data port Types1::Types2::Position {result1::result2 +=> constant access
       50.0};
flows
   flow1: flow sink signal {result1::result2 +=> constant access 50.0};
properties
   size => constant access +50.0;
end Thread3d;

thread Thread4
features
   Position_Input : in out parameter Types1::Types2::Position {result1::result2 => constant access
       50.0};
flows
   flow1: flow sink signal {result1::result2 +=> constant access 50.0};
properties
   size => constant access -50.0;
end Thread4;

thread Thread5
features
   Position_Input : in out event port;
flows
   flow1: flow sink signal {result1::result2 +=> constant access 50.0};
```

```
properties
    size => constant access 50.0;
end Thread5;

thread Thread6
features
    none;
flows
    none;
properties
    none;
end Thread6;
```

4.2 测试方式

操作步骤:

1. 将待测试文件命名为 'test.txt'文件，位置在源程序根目录下。
2. 运行 Java 文件包 'wordAnalysis'中的 'Main.java'文件。
3. 生成 'tokenOut.txt'文件作为语法分析的输入。
4. 运行 Java 文件包 'syntaxAnalysis'中的 'Main.java'文件。
5. 生成 'syntaxOut.txt'文件和 'syntaxErrorOut.txt'文件。
6. 其中 'syntaxOut.txt'中保存源文件对应的抽象语法树层次结构。
7. 'syntaxErrorOut.txt'中保存语法详细错误信息及其位置。
8. 注：在程序中，进行了两种错误处理，第一种是跳过错误项继续构建，同时抛出错误信息。第二种是遇见错误即停止程序，并在控制台报错。两种方式都会保存错误信息至 'syntaxErrorOut.txt'。此时采用第一种，若用第二种，则注释相应第一种代码，并将注释掉的退出程序代码恢复。

4.3 语法树构建结果

4.3.1 正确语法的语法树

```
|-Statement: THREAD_SPEC <IDENTIFIER1 , Thread1>
    |-Statement: FEATURE_SPEC <IDENTIFIER1 , AP_Position_Input>
        |-Statement: IO_TYPE <IN , in>
        |-Statement: PORT_SPEC
            |-Statement: PORT_TYPE <EVENT_DATA_PORT , event data port>
                |-Statement: REFERENCE <IDENTIFIER1 , Nav_Types> <IDENTIFIER2 , Position_GPS>
    |-Statement: FLOW_SPEC <IDENTIFIER1 , flow1>
        |-Statement: FLOW_PATH_SPEC <IDENTIFIER1 , signal   result1>
    |-Statement: ASSOCIATION <IDENTIFIER1 , dispatch_protocol>
        |-Statement: SPLITTER <EQUALTO , =>>
        |-Statement: DECIMAL <DECIMAL , 50.0>
```

```
|-Statement: THREAD_SPEC <IDENTIFIER1 , Thread_2>
    |-Statement: FEATURE_SPEC <IDENTIFIER1 , AP_Position_Input>
        |-Statement: IO_TYPE <IN , in>
        |-Statement: PORT_SPEC
            |-Statement: PORT_TYPE <DATA_PORT , data port>
                |-Statement: REFERENCE <IDENTIFIER1 , Nav_Types1> <IDENTIFIER2 , Nav_Types2> <IDENTIFIER3 , Position_GPS>
    |-Statement: FLOW_SPEC <IDENTIFIER1 , flow1>
        |-Statement: FLOW_SOURCE_SPEC <IDENTIFIER1 , signal>
            |-Statement: ASSOCIATION <IDENTIFIER1 , result1 result2>
                |-Statement: SPLITTER <PLUSEQUALTO , +=>>
                |-Statement: DECIMAL <DECIMAL , 50.0>
    |-Statement: ASSOCIATION <NONE , none>
```

```
|-Statement: THREAD_SPEC <IDENTIFIER1 , Thread3d>
    |-Statement: FEATURE_SPEC <IDENTIFIER1 , AP_Position_Input>
        |-Statement: IO_TYPE <OUT , out>
        |-Statement: PORT_SPEC
            |-Statement: PORT_TYPE <DATA_PORT , data port>
                |-Statement: REFERENCE <IDENTIFIER1 , Nav_Types1> <IDENTIFIER2 , Nav_Types2> <IDENTIFIER3 , Position_GPS>
            |-Statement: ASSOCIATION <IDENTIFIER1 , result1 result2>
                |-Statement: SPLITTER <PLUSEQUALTO , +=>>
                |-Statement: DECIMAL <DECIMAL , 50.0>
    |-Statement: FLOW_SPEC <IDENTIFIER1 , flow1>
        |-Statement: FLOW_SINK_SPEC <IDENTIFIER1 , signal>
            |-Statement: ASSOCIATION <IDENTIFIER1 , result1 result2>
                |-Statement: SPLITTER <PLUSEQUALTO , +=>>
                |-Statement: DECIMAL <DECIMAL , 50.0>
    |-Statement: ASSOCIATION <IDENTIFIER1 , size>
        |-Statement: SPLITTER <EQUALTO , =>>
        |-Statement: DECIMAL <DECIMAL , +50.0>
```

图 17: 语言样例语法树构建结果-1

```
|-Statement: THREAD_SPEC <IDENTIFIER1 , Thread4>
    |-Statement: FEATURE_SPEC <IDENTIFIER1 , AP_Position_Input>
        |-Statement: IO_TYPE <OUT , out>
        |-Statement: PARAMETER_SPEC
            |-Statement: REFERENCE <IDENTIFIER1 , Nav_Types1> <IDENTIFIER2 , Nav_Types2> <IDENTIFIER3 , Position_GPS>
            |-Statement: ASSOCIATION <IDENTIFIER1 , result1 result2>
                |-Statement: SPLITTER <EQUALTO , =>>
                |-Statement: DECIMAL <DECIMAL , 50.0>
    |-Statement: FLOW_SPEC <IDENTIFIER1 , flow1>
        |-Statement: FLOW_SINK_SPEC <IDENTIFIER1 , signal>
            |-Statement: ASSOCIATION <IDENTIFIER1 , result1 result2>
                |-Statement: SPLITTER <PLUSEQUALTO , +=>>
                |-Statement: DECIMAL <DECIMAL , 50.0>
    |-Statement: ASSOCIATION <IDENTIFIER1 , size>
        |-Statement: SPLITTER <EQUALTO , =>>
        |-Statement: DECIMAL <DECIMAL , -50.0>
```

```
|-Statement: THREAD_SPEC <IDENTIFIER1 , Thread5>
    |-Statement: FEATURE_SPEC <IDENTIFIER1 , AP_Position_Input>
        |-Statement: IO_TYPE <OUT , out>
        |-Statement: PORT_SPEC
            |-Statement: PORT_TYPE <EVENT_PORT , event port>
    |-Statement: FLOW_SPEC <IDENTIFIER1 , flow1>
        |-Statement: FLOW_SINK_SPEC <IDENTIFIER1 , signal>
            |-Statement: ASSOCIATION <IDENTIFIER1 , result1 result2>
                |-Statement: SPLITTER <PLUSEQUALTO , +=>>
                |-Statement: DECIMAL <DECIMAL , 50.0>
    |-Statement: ASSOCIATION <IDENTIFIER1 , size>
        |-Statement: SPLITTER <EQUALTO , =>>
        |-Statement: DECIMAL <DECIMAL , 50.0>
```

```
|-Statement: THREAD_SPEC <IDENTIFIER1 , Thread6>
    |-Statement: FEATURE_SPEC <NONE , none>
    |-Statement: FLOW_SPEC <NONE , none>
    |-Statement: ASSOCIATION <NONE , none>
```

图 18: 语言样例语法树构建结果-2

4.3.2 错误语法报错机制

```
Line 1: thread Thread1#
                      ^
    Error: invalid symbol '#'.(At Position: 15.)

Line 3:       _
              ^
    Error: invalid symbol '_'.(At Position: 2.)

Line 5:           1f
                   ^
    Error: invalid symbol 'f', there should be . or digit.(At Position: 10.)

Line 12:      _
              ^
    Error: invalid symbol '_'.(At Position: 2.)

Line 16:     dispatch_protocol => access .
                                         ^
    Error: invalid symbol '.'.(At Position: 30.)
```

图 19: 错误语法报错信息