Project 3 - Parsing

Parser.java

Recursive Descent Parser ... Using Lexer.java

ParserStarter.java

void scan ()
```
if nextToken != Token.EOF then
    nextToken = lexer.getToken ();
endIf
```
void syntaxError (msg)
```
Print message
Abort compiler
```
void mustHave (token, msg)
```
if nextToken == token then
    scan();
else
    syntaxError(msg);
endIf
```

© Harry H. Porter, 2005

Misc Details

Files in `../compilers/p3`
Create a new directory for each project
Two “tst” directories
- Files with no syntax errors:
  - tst/...
  - go, run, runAll
- Files with syntax errors:
  - tst2/...
  - go2, run2, runAll2

Lexer.class

Will use this when testing!!!
Use your own **Lexer.java**
- but test with my **Lexer.class**
- (Should make no difference)

Black Box: **Main.jar**
FTP
```
cd /pub/users/harry
    ⇒ ~harry/public_html/compilers
```

© Harry H. Porter, 2005

Main.java

```
p = new Parser (...);
p.parseProgram ();
```

Parser.java

```
class Parser {
  Lexer lexer;
  int nextToken;

  Parser (...) {...}
  void scan () {...}
  void syntaxError () {...}
  void mustHave () {...}
  void parseProgram () {...}
  ...
  void parseIfStmt () {...}
  ...
  void parseExpr () {...}
  ...
}
```

p

Parser

lexer

nextToken Token.INTEGER

Lexer

lineNumber 5

rValue

sValue

iValue 47

In Starter File

Later, we'll return Abstract Syntax Trees

© Harry H. Porter, 2005

For each non-terminal in the CFG...
Write a method

You may need to modify the grammar some...

Body → { Decl } **begin** { Stmt } **end**
...
Stmt → **if** Expr ...
→ **while** Expr ...
→ **write** WriteArgs ;
→ LValue := Expr ;
→ ID Arguments ;
→ ...
LValue → ID ...

Zero-or-more

© Harry H. Porter, 2005

```
Stmts   → Stmt Stmts
        → ε
Stmt    → if Expr ...
        → while Expr ...
        → write WriteArgs ;
        → IDStmt
        → ...
```

Parse zero-or-more "Stmt"s

```
method parseStmts()
  loop
    if nextToken == IF then
      parseIfStmt()
    elseIf nextToken == WHILE then
      parseWhileStmt()
    elseIf nextToken == WRITE then
      parseWriteStmt()
    elseIf nextToken == ID then
      parseIDStmt()
    ...
    else
      return
    endIf
  endLoop
endMethod
```

© Harry H. Porter, 2005


```
Body    → { Decl } begin { Stmt } end
```

```
method parseBody()
  parseDecls()
  mustHave(BEGIN, ...message...)
  parseStmts()
  mustHave(END, ...message...)
endMethod
```

Parses zero-or-more "Decl"s

Parse zero-or-more "Stmt"s

© Harry H. Porter, 2005

WriteArgs → "(" WriteExpr { , WriteExpr } ")"
→ "(" ")"

```
method parseWriteArgs()
  mustHave(LPAREN, "Expecting '('")
  if nextToken == RPAREN then
    scan()
    return
  endIf
  parseWriteExpr()
  while (nextToken == COMMA) do
    scan()
    parseWriteExpr()
  endWhile
  mustHave(RPAREN, "Expecting ',' or ')'")
endMethod
```

© Harry H. Porter, 2005

Statements That Start With ID

Problem:

Stmt → ...
→ LValue := Expr ;
→ ID Arguments ;
→ ...

Arguments → "(" Expr { , Expr } ")"
→ "(" ")"

Call Statement:

```
foo(x,y,z);
bar();
```

© Harry H. Porter, 2005

Statements That Start With ID

Problem:

```
Stmt       → ...
           → LValue  :=  Expr  ;
           → ID Arguments  ;
           → ...

Arguments  → "("  Expr  {  ,  Expr  }  ")"
           → "("  ")"
```

Call Statement:

```
foo(x,y,z);
bar();
```

Solution:

```
Stmt       → ...
           → IDStmt
           → ...

IDStmt     → ID  LValueMods  :=  Expr  ;
           → ID  "("  Arguments ;

Arguments  → Expr  {  ,  Expr  }  ")"
           → ")"
```

© Harry H. Porter, 2005

Creating Arrays in PCAT

```
type MyArr is array of integer;
var a: MyArr := nil;
...
a := MyArr  {  1, 1, 2, 3, 5, 8, 13  };
a := MyArr  {  1000 of -1  };
a := MyArr  {  300 of -1, 150 of -2, -3, -4, x+y  };
```

© Harry H. Porter, 2005

Creating Arrays in PCAT

```
type MyArr is array of integer;
var a: MyArr := nil;
...
a := MyArr  {{  1, 1, 2, 3, 5, 8, 13  }};
a := MyArr  {{  1000 of -1  }};
a := MyArr  {{  300 of -1, 150 of -2, -3, -4, x+y  }};
```

A change to the PCAT language (see e-mail)

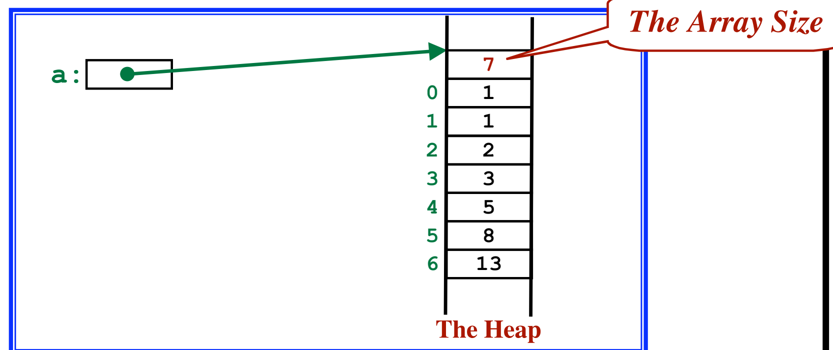
© Harry H. Porter, 2005

Creating Arrays in PCAT

```
type MyArr is array of integer;
var a: MyArr := nil;
...
a := MyArr  {{  1, 1, 2, 3, 5, 8, 13  }};
a := MyArr  {{  1000 of -1  }};
a := MyArr  {{  300 of -1, 150 of -2, -3, -4, x+y  }};
```

Runtime View:

Arrays are stored in the heap!
All array variables are 1 word in size.



© Harry H. Porter, 2005

Creating Arrays in PCAT

```
type MyArr is array of integer;
var a: MyArr := nil;
...
a := MyArr  {{  1, 1, 2, 3, 5, 8, 13  }};
a := MyArr  {{  1000 of -1  }};
a := MyArr  {{  300 of -1, 150 of -2, -3, -4, x+y  }};
```

Problem:

ArrayValues → '{' '{' ArrayValue { ',' ArrayValue } '}' '}'

ArrayValue → [Expr **of**] Expr

Count Expression (optional)

Value Expression (not optional)

Creating Arrays in PCAT

```
type MyArr is array of integer;
var a: MyArr := nil;
...
a := MyArr  {{  1, 1, 2, 3, 5, 8, 13  }};
a := MyArr  {{  1000 of -1  }};
a := MyArr  {{  300 of -1, 150 of -2, -3, -4, x+y  }};
```

Problem:

ArrayValues → '{' '{' ArrayValue { ',' ArrayValue } '}' '}'

ArrayValue → [Expr **of**] Expr

Count Expression (optional)

Value Expression (not optional)

$Expr_{VAL}$

$Expr_{CNT}$ **of** $Expr_{VAL}$

Creating Arrays in PCAT

```
type MyArr is array of integer;
var a: MyArr := nil;
...
a := MyArr  {{  1, 1, 2, 3, 5, 8, 13  }};
a := MyArr  {{  1000 of -1  }};
a := MyArr  {{  300 of -1, 150 of -2, -3, -4, x+y  }};
```

Problem:

ArrayValues → '{' '{' ArrayValue { ',' ArrayValue } '}' '}'
ArrayValue → [Expr **of**] Expr

Count Expression (optional)

Value Expression (not optional)

$\text{Expr}_{\textbf{VAL}}$
$\text{Expr}_{\textbf{CNT}}$ **of** $\text{Expr}_{\textbf{VAL}}$

Solution:

ArrayValue → Expr [**of** Expr]

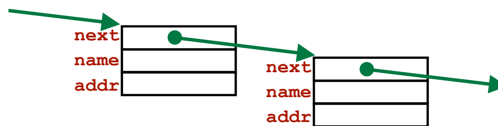
© Harry H. Porter, 2005

L-Values in PCAT

LValue → ID
→ LValue **[** Expr **]** — *Array References*
→ LValue **.** ID — *Field Accessing in Records*

Examples:

```
x
x[4]
x[4][j+1][k*foo(y)]
x.name
x.next.next.next
x.next.addr[3].street
```



© Harry H. Porter, 2005

L-Values in PCAT

```
LValue  → ID
        → LValue [ Expr ]
        → LValue . ID
```

Array References (LValue [Expr])

Field Accessing in Records (LValue . ID)

Examples:

```
x
x[4]
x[4][j+1][k*foo(y)]
x.name
x.next.next.next
x.next.addr[3].street
```


Solution:

```
LValue      → ID  LValueMods
LValueMods  → { [ Expr ] | . ID }
```


Expressions in PCAT

```
Expr      →  INTEGER
          →  REAL
          →  LValue
          →  "(" Expr ")"
          →  UnaryOp Expr
          →  Expr BinaryOp Expr
          →  ID Arguments
          →  ID FieldInits
          →  ID ArrayValues
UnaryOp   →  + | - | not
BinaryOp  →  + | - | * | / | div | mod | or | and
          →  < | <= | > | >= | = | <>
```

Examples:

```
47
123.56
x
x.field
x[4]
-(x+(y*4-k))
foo(...)
MyRec{f1:=100; f2:=x}
MyArr{1,1,2,3,5,6}
```

"All Binary ops are left-associative.
Precedence should be..."

```
foo(...)   (Expr)            ← Highest
unary+   unary-   NOT
*   /   MOD   DIV   AND
+   -   OR
<   <=   >   >=   =   <>     ← Lowest
```

Solution

Expr → Expr2 { (**<** | **<=** | **>** | **>=** | **=** | **<>**) Expr2 }
Expr2 → Expr3 { (**+** | **-** | **or**) Expr3 }
Expr3 → Expr4 { (***** | **/** | **mod** | **div** | **and**) Expr4 }
Expr4 → **+** Expr4 | **-** Expr4 | **not** Expr4 | Expr5
Expr5 → "**(**" Expr "**)**"
→ INTEGER
→ REAL
→ IDMods
IDMods → ID Arguments
→ ID ArrayValues
→ ID FieldInits
→ ID LValueMods

We know the FIRSTs of these:
(
{ {
{ *< anything besides '{' >*
< anything else>

Methods:

```
parseExpr ()
parseExpr2 ()
parseExpr3 ()
parseExpr4 ()
parseExpr5 ()
parseIDMods ()
```


Checking Your Parser

Add "print" statements

Source: ... `x := ( 1 + y ) div ( 2.3 < z ) ;` ...

Output:

```
...
x
ASSIGN
1
y
PLUS
2.3
z
LESS
DIV
ENDASSIGN
```

Error Recovery:

Very hard!

... Do the basic assignment first!

Questions:

ASK!

Project 4: Build the AST

Modify methods to build and return AST

REQUIRES Project 3 to be working!