PCAT Delta
Porter's Version vs. Tolmach's Version

Harry Porter
January 21, 2006

Introduction

There are several small differences between the PCAT language used in Andrew Tolmach's Compiler course and the version used in my class. These differences are discussed here.

The abbreviation "T&L" refers to Tolmach and Li's language, as documented in [*The PCAT Programming Language Reference Manual*, Andrew Tolmach and Jingke Li, revised October 8, 2004].

Uppercase/Lowercase Keywords

The keywords in T&L are all uppercase. Porter uses the same keywords, but they are lowercase.

T&L	**Porter**
`BEGIN`	`begin`
`PROCEDURE`	`procedure`
etc.	

This difference is purely lexical and will not affect the back-end.

The "ELSIF" Keyword

There is one difference in the set of keywords. T&L spells the following keyword differently:

T&L	**Porter**
`ELSIF`	`elseif`

This difference is purely lexical and will not affect the back-end.

Punctuation for Array Initialization

T&L uses braces for array initialization. Porter uses [< and >] for array initialization.

Here is an example:

T&L

```
VAR a: MyArray := MyArray { 7, 9, 3 OF 11, 13, 15 };
```

Porter

```
var a: MyArray := MyArray {{ 7, 9, 3 of 11, 13, 15 }};
```

In Porter, there are two lexical tokens, not in T&L. Each of these tokens, "{{" and "}}", consists of two characters, somewhat like the := and <> tokens used for assignment and not-equal.

Punctuation Tokens in Porter

```
:=
+
-
*
/
<
<=
>
>=
=
<>
:
;
,
.
)
(
[
]
{
}
{{          ← Not in T&L
}}          ← Not in T&L
```

This difference is purely lexical and will not affect the back-end.

Associativity of Relational Operators

In T&L, the relational operators are not associative, while in Porter they are left-associative.

T&L

```
(a = b) = c
 a = b  = c          ← Syntax error
```

Porter

```
(a = b) = c
 a = b  = c          ← Okay; means the same
```

This makes a slight difference when using expressions involving = (equal) and <> (not equal), but note that in the case of the comparison operators (<, <=, >=, >), expressions like

```
 a <  b  < c
(a <  b) < c
 a < (b  < c)
```

are semantically incorrect anyway.

This difference is purely grammatical and will not affect the back-end.

Recursive Types

Consider two mutually recursive type T1 and T2. T&L uses the keyword AND, while Porter does not.

T&L

```
TYPE
    T1 IS RECORD
             val: INTEGER;
             next: T2;
          END;
  AND
    T2 IS RECORD
             val: INTEGER;
             next: T1;
          END;
```

Porter

```
type
    T1 is record
             val: integer;
             next: T2;
          end;
    T2 is record
             val: integer;
             next: T1;
          end;
```

In Porter, a single body may have several "type" declarations and each may define several type names. Any type name may be defined in terms of any other type name in the same body.

For example:

Porter

```
type
    S1 is record
             val: integer;
             next: S2;
          end;
    S2 is record
             val: integer;
             next: S3;
          end;
type
    S3 is record
             val: integer;
             next: S4;
          end;
    S4 is record
             val: integer;
             next: S1;
          end;
```

is semantically equivalent to:

T&L

```
TYPE
    S1 IS RECORD
             val: INTEGER;
             next: S2;
          END;
  AND
    S2 IS RECORD
             val: INTEGER;
             next: S3;
          END;
  AND
    S3 IS RECORD
             val: INTEGER;
             next: S4;
          END;
  AND
    S4 IS RECORD
             val: INTEGER;
             next: S1;
          END;
```

In the Abstract Syntax Tree representation used by Porter, the grouping of type declarations is lost and the following two examples would result in the same internal representation:

Source Code #1

```
type
  T1 is record
          val: integer;
          next: T2;
        end;
  T2 is record
          val: integer;
          next: T1;
        end;
```

Source Code #2

```
type
  T1 is record
          val: integer;
          next: T2;
        end;
type
  T2 is record
          val: integer;
          next: T1
        end;
```

Recursive Procedures

A similar situation exists when recursive procedures are defined. In T&L the AND keyword is used, while it is implicit in Porter.

T&L

```
PROCEDURE
    foo () IS BEGIN ... bar() ... END;
  AND
    bar () IS BEGIN ... foo() ... END;
```

Porter

```
procedure
    foo () is begin ... bar() ... end;
    bar () is begin ... foo() ... end;
```

In Porter, the "and" is implicit and procedures can be mutually recursive whenever they are defined in the same body. Consequently, in Porter, the "procedure" keyword is often repeated; the above example would often be coded as:

Porter

```
procedure foo () is begin ... bar() ... end;
procedure bar () is begin ... foo() ... end;
```

In the Abstract Syntax Tree representation used by Porter, the grouping of procedures is lost and the following two examples would result in the same internal representation:

Source Code #1

```
procedure
    foo () is begin ... bar() ... end;
    bar () is begin ... foo() ... end;
```

Source Code #2

```
procedure foo () is begin ... bar() ... end;
procedure bar () is begin ... foo() ... end;
```

Repetition of VarDecls, TypeDecls, and ProcDecls

In Porter, the grammar for declarations allows one-or-more occurrences; the rules are:

```
Declaration      → var VarDecl { VarDecl }
                 → type TypeDecl { TypeDecl }
                 → procedure ProcedureDecl { ProcedureDecl }
```

In T&L, the grammar rules are slightly different, but effectively generate the same thing:

```
declaration      → VAR var-decls
                 → TYPE type-decls
                 → PROCEDURE procedure-decls

var-decls        → var-decl { var-decl }
type-decls       → type-decl { type-decl }
procedure-decls  → procedure-decl { AND procedure-decl }
```

Terminology

The non-terminals in the T&L grammar are similar to the non-terminals used in Porter, but there are a few minor differences in spelling.

T&L	Porter
program	Program
body	Body
declaration	Declaration
var-decls	
var-decl	VarDecl
type-decls	
type-decl	TypeDecl
procedure-decls	
procedure-decl	ProcedureDecl
typename	TypeName
type	CompoundType
component	FieldDecl
formal-params	FormalParams
fp-section	FormalSection
statement	Statement
write-params	WriteArgs
write-expr	WriteExpr
expression	Expression
lvalue	LValue
actual-params	Arguments
record-inits	FieldInits
array-inits	ArrayValues
array-init	ArrayValue
number	Number
unary-op	UnaryOp
binary-op	BinaryOp

Porter refers to the components in a record as “fields” while T&L refers to the same concept as “components.”

Porter refers to the initializing expressions in a record as “FieldInits” while T&L refers to the same concept as “record-inits.”

Porter uses the term “arguments”, while T&L uses the term “actual-params.”