PL_TDF Definition

Issue 2.1.0 (Aug 1993)

Crown Copyright ©June 1993

TABLE OF CONTENTS

1 Introduction . . . 1

2 Notation . . . 2

2.1 Syntax description . . . 2

2.2 Lexical Units . . . 2

2.3 Pre-processing . . . 3

3 The Language . . . 4

3.1 Program . . . 5

3.1.1 Tokdec . . . 5

3.1.2 Tokdef . . . 5

3.1.3 Tagdec . . . 6

3.1.4 Tagdef . . . 6

3.1.5 Altagdef . . . 6

3.1.6 Structdef . . . 7

3.1.7 Procdef . . . 7

3.2 First-class SORT expansions . . . 8

3.2.1 Access . . . 8

3.2.2 Al_tag . . . 8

3.2.3 Alignment . . . 8

3.2.4 Bitfield_variety . . . 9

3.2.5 Bool . . . 9

3.2.6 Error_treatment . . . 9

3.2.7 Exp . . . 9

3.2.7.1 ExpTerm . . . 10

3.2.8 Floating_variety . . . 11

3.2.9 Label . . . 11

3.2.10 Nat . . . 12

3.2.11 Ntest . . . 12

3.2.12 Rounding_mode . . . 13

3.2.13 Shape . . . 13

3.2.14 Signed_Nat . . . 13

3.2.15 Tag . . . 14

3.2.16 Token . . . 14

3.2.17 Transfer_mode . . . 14

3.2.18 Variety . . . 14

3.3 Control structure and local declarations . . . 15

3.3.1 ConditionalExp and Assertion . . . 15

3.3.2 RepeatExp . . . 16

3.3.3 LabelledExp . . . 16

3.3.4 Local_Defn . . . 16

4 Example PL_TDF programs . . . 17

4.1 Sieve of Erastothenes . . . 17

4.2 Example with structures . . . 19

4.3 Test for case . . . 20

4.4 Example of use of high-order TOKENs . . . 21

4.5 A test for long jumps . . . 22

5 Use of the PL_TDF compiler . . . 23

1 Introduction

PL_TDF is a language in the lineage of Wirth's PL360 and its later derivatives. The basic idea in PL360 was to give one an assembler in which one could express all of the order-code of the IBM 360 while still preserving the logical structure of the program using familiar programming constructs. If one had to produce a program at the code level, this approach was much preferable to writing "flat" assembly code using a traditional assembler, as anyone who has used both can testify.

In the TDF "machine" the problem is not lack of structure at its "assembly" level, but rather too much of it; one loses the sense of a TDF program because of its deeply nested structure. Also the naming conventions of TDF are designed to make them tractable to machine manipulation, rather than human reading and writing. However, the approach is basically the same. PL_TDF provides shorthand notations for the commonly occuring control structures and operations while still allowing one to use the standard TDF constructors which, in turn, may have shorthand notations for their parameters. The naming is always done by identifiers where the sort of the name is determined by its declaration, or by context.

The TDF derived from PL_TDF is guaranteed to be SORT correct; however, there is no SHAPE checking, so one can still make illegal TDF.

2 Notation

2.1 Syntax description

Words enclosed in angle brackets, < >, form non-terminal symbols. Other symbols and words stand for themselves as terminal symbols. An expansion of a non-terminal is indicated using ::= with its expansion given as a sequence (possibly empty) of terminals and non-terminals. For example:

<Exp> ::= * <ident>

is a possible expansion of an EXP SORT. If the word for the non-terminal starts with a capital letter then it will be totally described by a set of such expansions; otherwise the expansion of the non-terminal will be given by other methods in the text.

The post-fix -Opt on a non terminal is an abreviation allowing an empty expansion. For example:

<Access>-Opt

is equivalent to the use of another non-terminal <AccessOption> whose expansions are:

<AccessOption> ::=

<AccessOption> ::= <Access>

The post-fix -List on a non terminal is an abreviation for lists of objects seperated by the ,-symbol. For example:

<Exp>-List

is equivalent to the use of another non-terminal <ExpList> whose expansions are:

<ExpList> ::= <Exp>

<ExpList> ::= <ExpList> , <Exp>

Both of these post-fix notations are also used with sequences of terminals and non-terminals within the angle brackets with the same kind of expansion. In these cases, the expansion within the angle brackets form an anonymous non-terminal.

2.2 Lexical Units

The terminal symbols ( ), [ ], and { } always occur as parenthetic pairs and never form part of other terminal symbols.

The terminal symbols , ; and : are similarly terminators for other terminal symbols.

White space is a terminator for other terminal symbols but is otherwise ignored except in strings.

All other terminal symbols are sequences of ACSII symbols not including the above. These are divided into seven classes: keywords, TDF constructors, operators, <integer_denotation>s, <floating_denotation>s, <string>s and <ident>s.

The keywords and operators are expressed directly in the syntax description. The TDF constructors are those given in the TDF specification which have first-class SORTs as parameters and results.

An <integer_denotatation> allows one to express an integer in any base less than 16, with the default being 10.

<integer_denotation> ::= <digit>
<integer_denotation> ::= <integer_denotation> <digit>
<integer_denotation> ::= <base> <integer_denotation>

<base> :: = <integer_denotation> r

Examples are 31, 16r1f, 8r37, 2r11111 - all giving the same value.

A <floating_denotation> is an <integer_denotation> followed by the . symbol and a sequence of digits. The radix of the <floating_denotation> is given by the base of its component <integer_denotation>

A <string> is the same as a C string - any sequence of characters within " ". The same C conventions hold for \ within strings for single characters.

A <character> is an string character within ' '. The same \ conventions hold.

An <ident> is any other sequence of characters. They will be used to form names for TAGs, TOKENs, AL_TAGs and LABELs.

2.3 Pre-processing

At the moment there is only one pre-processing directive. A line starting with #include will textually include the following file (named within string quotes), using the same path conventions as C.

Comments may be included in the text using the /* ... */ notation; this differs slightly from the C convention in that comments may be nested.

3 The Language

The basic philosophy of PL_TDF is to provide the "glue" constructors of TDF automatically, while still allowing the programmer to use the significant constructors in their most general form. By "glue" constructors, I mean those like make_link, make_group etc. which are there to provide tedious, but vital, constructions concerned with linking and naming. The "significant" constructors really come in two groups, depending on their resulting SORTs. There are those SORTs like TOKDEC, whose SORTs are purely syntactic and can't be used as results of token applications or _cond constructions. On the other hand, the first-class SORTs, like EXP, can be used in those situations and generally have a much richer set of constructors. These first-class SORTs are precisely those which have SORTNAMEs. These SORTNAMEs appear in PL_TDF as expansions of <Sortname>:

<Sortname> :: = ACCESS
<Sortname> :: = AL_TAG
<Sortname> :: = ALIGNMENT
<Sortname> :: = BITFIELD_VARIETY
<Sortname> :: = BOOL
<Sortname> :: = ERROR_TREATMENT
<Sortname> :: =EXP
<Sortname> :: =FLOATING_VARIETY
<Sortname> :: =LABEL
<Sortname> :: = NAT
<Sortname> :: = NTEST
<Sortname> ::= ROUNDING_MODE
<Sortname> ::= SHAPE
<Sortname> ::= SIGNED_NAT
<Sortname> ::= TAG
<Sortname> ::= TRANSFER_MODE
<Sortname> ::= VARIETY

All of the significant constructors are expanded by non-terminals with names related to their resulting SORT e.g. all EXPs are expanded by <Exp> and all TOKDECs are expanded by <Tokdec>. Any first-class SORT can be expanded by using the constructor names given in the TDF specification, provided that the parameter SORTs are also first-class. For example, the following are all valid expansions of <Exp> :

make_top	
return(E)	where E is an expansion of <Exp>
goto(L)	where L is an expansion of <Label>
assign(E1, E2)	where E1 and E2 are expansions of <Exp>

Any such use of TDF constructors will be checked for the SORT-correctness of their parameters. I will denote such a constructor as an <exp_constructor>; similarly for all the other first-class sorts.

Any of the first-class sorts may also be expanded by a token application. Tokens in PL_TDF are given <ident> names by <Tokdef> or <Tokdec> which must occur before their use in applications. In applications, these names will be denoted by <exp_token>, <shape_token> etc. , depending on the result sort of their introduction.

The principle of "no use before declaration" also applies to <ident> names given to TAGs.

3.1 Program

The root expansion of a PL_TDF program is given by <Program>:

<Program> ::= <ElementList> Keep (<Item>-List-Opt)

<ElementList> ::= <Element> ;
<ElementList> ::= <Element> ; <ElementList>

<Element> ::= <Tokdec>
<Element> ::= <Tokdef>
<Element> ::= <Tagdec>
<Element> ::= <Tagdef>
<Element> ::= <Altagdef>
<Element> ::= <Structdef>
<Element> ::= <Procdef>

<Item> :: = <tag>
<Item> ::= <token>
<item> ::= <altag>

A <Program> consists of a list of definitions and declarations giving meaning to various <ident>s, as TAGs, TOKENs and AL_TAGs. The <Item>-List-Opt indicates which of these names will be externally available via CAPSULE_LINKs; in addition any other names which are declared but not defined will also be linked externally.

A <Program> will produce a single TDF CAPSULE.

3.1.1 Tokdec

A <Tokdec> introduces an <ident> as a TOKEN:

<Tokdec> ::= Tokdec <ident> : [<TokDecPar>-List-Opt] <ResultSort>

<ResultSort> ::= <Sortname>

<TokDecPar> ::= <Sortname>
<TokDecPar> ::= TOKEN [<TokDecPar>-List-Opt] <ResultSort>

This produces a TOKDEC in a tokdec UNIT of the CAPSULE. Further uses of the introduced <ident> will be treated as a <x-token> where x is given by the <ResultSort>.

3.1.2 Tokdef

A <Tokdef> defines an <ident> as a TOKEN; this <ident> may have previously been introduced by a <Tokdec>:

<Tokdef> ::= Tokdef <ident> = <Tok_Defn>

<Tok_Defn> ::= [<TokDefPar>-List-Opt] <ResultSort> <result_sort>

<TokDefPar> ::= <ident> : <TokDecPar>

This produces a TOKDEF in a tokdef UNIT of the CAPSULE. The expansion of <result_sort> depends on <ResultSort>, e.g. if <ResultSort> is EXP then <result_sort> ::= <Exp> and so on.

Each of the <ident>s in the <TokDefPar>s will be names for tokens whose scope is <result_sort>. A use of such a name within its scope will be expanded as a parameterless token application of the appropriate sort given by its <TokDecPar>. Note that this is still true if the <TokDecPar> is a TOKEN - if a <TokDefPar> is:

x: TOKEN[LABEL]EXP

then x[L] is expanded as:

exp_apply_token(token_apply_token(x, ()), L)

<Tok_defn> also occurs in an expansion of <Token>, as a parameter of a token application.

3.1.3 Tagdec

A <Tagdec> introduces an <ident> as a TAG:

<Tagdec> ::= <DecType> <ident> <Access>-Opt : <Shape>

<DecType> ::= Vardec
<DecType> ::= Iddec
<DecType> ::= Commondec

This produces a TAGDEC in a tagdec UNIT of the CAPSULE, using a make_id_tagdec for the Iddec option, a make_var_tagdec for the Vardec option and a common_tagdec for the Commondec option.

The <Shape>s in both <Tagdec>s and <Tagdef>s will produce SHAPE TOKENs in a tagdef UNIT; these may be applied in various shorthand operations on TAG <ident>s.

3.1.4 Tagdef

A <Tagdef> defines an <ident> as a TAG. This <ident> may have previously been introduced by a <Tagdec>; if it has not the < : <Shape> >-Opt below must not be empty and a TAGDEC will be produced for it.

<Tagdef> ::= Var <ident> < : <Shape> >-Opt < = <Exp>>-Opt

Produces a make_var_tagdef.

<Tagdef> ::= Common <ident> < : <Shape> >-Opt < = <Exp> >-Opt

Produces a common_tagdef.

<Tagdef> ::= Let <ident> < : <Shape> >-Opt = <Exp>

Produces a make_id_tagdef.

<Tagdef> ::= String <ident> <Variety>-Opt =<string>

This is a shorthand for producing names which have the properties of C strings. The <Variety>-Opt gives the variety of the characters with the string, an empty option giving unsigned chars. The TDF produced is a make_ var_tagdef initialised by a make_nof_int. This means that given a String definition:

String format = “Result = %d\n”

the tag <ident>, format, could be used straightforwardly as the first parameter of printf - see Section 4 (Example PL_TDF programs).

3.1.5 Altagdef

An <Altagdef> defines an <ident> as an AL_TAG:

<Altagdef> ::= Al_tagdef <ident> = <Alignment>

This produces an AL_TAGDEF in an al_tagdef UNIT of the CAPSULE. The <ident> concerned can be previously used in as an expansion of <Alignment>.

3.1.6 Structdef

A <Structdef> defines a TOKEN for a structure SHAPE, together with two TOKENs for each field of the structure to allow easy access to the offsets and contents of the field:

<Structdef> ::= Struct <Structname> (<Field>-List)

<Structname> ::= <ident>

<Field> ::= <Fieldname> : <Shape>

<Fieldname> ::= <ident>

This produces a TOKDEF in a tokdef UNIT defining <Structname> as a SHAPE token whose expansion is an EXP OFFSET(a1,a2) where the OFFSET is the size of the structure with standard TDF padding and offset addition of the component SHAPEs and sizes[1].

Each <Fieldname> will produce two TOKENs. The first is named by <Fieldname> itself and is a [EXP]EXP which gives the value of the field of its structure parameter. The second is named by prefixing <Fieldname> by the .-symbol and is an [ ]EXP giving the OFFSET of the field from the start of the structure. Thus given:

Struct Complex (re: Double, im: Double)

Complex is a TOKEN for a SHAPE defining two Doubles; re[E] and im[E] will extract the components of E where E is an EXP of shape Complex; .re and .im give EXP OFFSETs of the the two fields from the start of the structure.

3.1.7 Procdef

A <Procdef> defines a TAG to be a procedure; it is simply an abreviation of a an Iddec <Tagdef>:

<Procdef> ::= Proc <ident> = <Proc_Defn>

<Proc_defn> ::= <Shape> (<TagShAcc>-List-Opt <VarIntro>-Opt) <ClosedExp>

<TagShAcc> ::= <Parametername> <Access>-Opt : <Shape>

<Parametername> ::= <ident>

<VarIntro> ::= Varpar <Varparname> : <Alignment>

<Varparname> ::= <ident>

A <Procdef> produces a TAGDEF in a tagdef UNIT and and, possibly, a TAGDEC in a tagdef UNIT. A <Proc_Defn> produces a make_proc with the obvious operands. The scope of the tag names introduced by <Parametername> and <Varparname> is the <ClosedExp> (see section 3.3 on page 15).

1. Note that this may not correspond precisely with C sizes.

3.2 First-class SORT expansions

All of the first-class sorts have similar expansions for native TDF constructions and for token applications. I shall take <Shape> as the paradigm sort and allow the reader to conjugate the following for the the other sorts.

Those first-class sorts which include the _cond constructions denote them in the same way:

<Shape> ::= SHAPE ? (<Exp>, <Shape>, <Shape>)

This produces a shape_cond with the obvious parameters.

Each constructor for <Shape> with parameters which are first-class sorts can be expanded:

<Shape> :: = <shape_constructor> < (<constructor_param>-List) >-Opt

Each <constructor_param> will be the first_class SORT expansion, required by the <shape_constructor> as in the TDF specification eg the constructor, pointer, requires a <constructor_param> ::= <Alignment> .

Any <ident> which is declared to be a <shape_token> by a TOKDEF or TOKDEC can be expanded:

<Shape> ::= <shape_token> < [<token_param>-List] >-Opt

This will produce a shape_apply_token with the appropriate parameters. Each <token_param> will be the first-class SORT expansion required by the SORT given by the <TokDecPar> of the TOKDEF or TOKDEC which introduced <shape_token>.

3.2.1 Access

<Access> ::= ACCESS ? (<Exp> , <Access> , <Access>)

<Access> ::= <access_constructor> < (<constructor_param>-List) >-Opt

<Access> ::= <access_token> < [<token_param>-List] >-Opt

There are no expansions of <Access> other than the standard ones.

3.2.2 Al_tag

<Al_tag> ::= <al_tag_token> < [<token_param>-List] >-Opt

The standard token expansion.

<Al_tag> ::= <ident>

Any <ident> found as an expansion of <Al_tag> will be declared as the name for an AL_TAG.

3.2.3 Alignment

<Alignment> ::= ALIGNMENT ? (<Exp> , <Alignment> , <Alignment>)

<Alignment> ::= <alignment_constructor> < (<constructor_param>-List) >-Opt

<Alignment> ::= <alignment_token> < [<token_param>-List] >-Opt

The standard expansions.

<Alignment> ::= <Al_tag>

This results in an obtain_al_tag of the AL_TAG.

<Alignment> ::= (<Alignment>-List-Opt)

The <Alignment>s in the <Alignment>-List are united using unite_alignments. The empty option results in the top ALIGNMENT.

3.2.4 Bitfield_variety

<Bitfield_variety> ::= BITFIELD_VARIETY ? (<Exp> , <Bitfield_variety>, <Bitfield_variety>)
<Bitfield_variety> ::= <bitfield_variety_constructor> < (<constructor_param>-List) >-Opt
<Bitfield_variety> ::= <bitfield_variety__token> < [<token_param>-List] >-Opt

The standard expansions.

<Bitfield_variety> ::= <BfSign>-Opt <Nat>

<BfSign> ::= <Bool>
<BfSign> ::= Signed
<BfSign> ::= Unsigned

This expands to bfvar_bits. The empty default on the sign is Signed.

3.2.5 Bool

<Bool> ::= BOOL ? (<Exp> , <Bool>, <Bool>)
<Bool> ::= <bool_constructor> < (<constructor_param>-List) >-Opt
<Bool> ::= <bool_token> < [<token_param>-List] >-Opt

There are no expansions of <Bool> other than the standard ones.

3.2.6 Error_treatment

<Error_treatment> ::= ERROR_TREATMENT ?
(<Exp> , <Error_treatment>, <Error_treatment>)
<Error_treatment> ::= <error_treatment_constructor> < (<constructor_param>-List) >-Opt
<Error_treatment> ::= <error_treatment__token> < [<token_param>-List] >-Opt

The standard expansions.

<Error_treatment> ::= <Label>

This gives an error_jump to the label.

3.2.7 Exp

<Exp> ::= <ExpTerm>
<Exp> ::= <ExpTerm> <BinaryOp> <ExpTerm>

The <BinaryOp>s include the arithmetic, offset, logical operators and assignment and are given in table 1. In this expansion, any error_treatments are taken to be wrap.

Table 1.

<BinaryOp>	TDF constructor		<BinaryOp>	TDF constructor
And	and		Or	or
Xor	xor		*+.	add_to_ptr
-	subtract_ptrs		.*	offset_mult
.+.	offset_add		.-.	offset_subtract
./	offset_div_by_int		./.	offset_div
.max.	offset_max		%	rem2
%1	rem1		*	mult
+	plus		-	minus

Table 1.

<BinaryOp>	TDF constructor	<BinaryOp>	TDF constructor
/	div2	/1	div1
<<	shift_left	>>	shift_right
F*	floating_mult	F+	floating_plus
F-	floating_minus	F/	floating_div
=	assign		

The names like *+. (i.e. add_to_ptr) do have a certain logic; the * indicates that the left operand must be a pointer expression and the . that the other is an offset

The further expansions of <Exp> are all <ExpTerm>s

3.2.7.1 ExpTerm

<ExpTerm> ::= EXP ? (<Exp> , <Exp>, <Exp>)

<ExpTerm> ::= <exp_constructor> < (<constructor_param>-List) >-Opt

<ExpTerm> ::= <exp_token> < [<token_param>-List] >-Opt

The standard expansions.

<ExpTerm> ::= <ClosedExp>

For <ClosedExp>, see section 3.3 on page 15.

<ExpTerm> ::= (<Exp>)

<ExpTerm> ::= - (<Exp>)

The negate constructor.

<ExpTerm> ::= Sizeof (<Shape>)

This produces the EXP OFFSET for an index multiplier for arrays of <Shape>. It is the shape_offset of <Shape> padded up to its alignment.

<ExpTerm> ::= <Tag>

This produces an obtain_tag.

<ExpTerm> ::= * <ident>

The <ident> must have been declared as a variable TAG and the construction produces a contents operation with its declared SHAPE.

<ExpTerm> ::= * (<Shape>) <ExpTerm>

This produces a contents operation with the given <Shape>.

<ExpTerm> ::= <Assertion>

For <Assertion>, see section 3.3.1 on page 15

<ExpTerm> ::= Case <Exp> (<RangeDest>-List)

<RangeDest> ::= <Signed_Nat> < : <Signed_Nat> >-Opt -> <Label>

This produces a case operation.

<ExpTerm> ::= Cons [<Exp>] (< <Offset> : <Exp> >-List)

<Offset> ::= <Exp>

This produces a make_compound with the [<Exp>] as the size and fields given by < <Offset> : <Exp> >-List.

<ExpTerm> ::= [<Variety>] <ExpTerm>

This produces a change_variety with a wrap error_treatment.

<ExpTerm> ::= <Signed_Nat> (<Variety>)

This produces a make_int of the <Signed_Nat> with the given variety.

<ExpTerm> ::= <floating_denotation> < E <Signed_Nat> >-Opt
<ExpTerm> ::= - <floating_denotation> < E <Signed_Nat> >-Opt

Produces a make_floating.

<ExpTerm> ::= <ProcVal> [<Shape>] (<Exp>-List-Opt < Varpar <Exp> >-Opt)

<ProcVal> :: = <Tag>
<ProcVal> :: = (<Exp>)

Produces an apply_proc with the given parameters returning the given <Shape>.

<ExpTerm> ::= Proc <Proc_defn>

Produces a make_proc. For <Proc_defn>, see section 3.1.7 on page 8

<ExpTerm> ::= <string> (<Variety>)

Produces a make_nof_int of the given variety.

<ExpTerm> ::= # <string>

This produces a TDF fail_installer; this construction is useful for narrowing down SHAPE errors detected by the translator.

3.2.8 Floating_variety

<Floating_variety> ::= FLOATING_VARIETY ?
(<Exp> , <Floating_variety>, <Floating_variety>)
<Floating_variety> ::= <floating_variety_constructor> < (<constructor_param>-List) >-Opt
<Floating_variety> ::= <floating_variety__token> < [<token_param>-List] >-Opt

The standard constructions.

<Floating_variety> ::= Float

An IEEE 32 bit floating variety.

<Floating_variety> ::= Double

An IEEE 64 bit floating variety.

3.2.9 Label

<Label> ::= <label_token> < [<token_param>-List] >-Opt

The standard token application.

<Label> ::= <ident>

The <ident> will be declared as a LABEL, whose scope is the current procedure.

3.2.10 Nat

<Nat> ::= NAT ? (<Exp> , <Nat>, <Nat>)

<Nat> ::= <nat_constructor> < (<constructor_param>-List) >-Opt

<Nat> ::= <nat_token> < [<token_param>-List] >-Opt

The standard expansions.

<Nat> ::= <integer_denotation>

Produces a make_nat on the integer

<Nat> ::= <character>

Produces a make_nat on the ASCII value of the character.

3.2.11 Ntest

<Ntest> ::= NTEST ? (<Exp> , <Ntest>, <Ntest>)

<Ntest> ::= <ntest_constructor> < (<constructor_param>-List) >-Opt

<Ntest> ::= <ntest_token> < [<token_param>-List] >-Opt

The standard expansions.

<Ntest> ::= !<

Produces not_less_than.

<Ntest> ::= !<=

Produces not_less_than_or_equal.

<Ntest> ::= !=

Produces not_equal.

<Ntest> ::= !>

Produces not_greater_than.

<Ntest> ::= !>=

Produces not_greater_than_or_equal.

<Ntest> ::= !Comparable

Produces not_comparable.

<Ntest> ::= <

Produces less_than.

<Ntest> ::= <=

Produces less_than_or_equal.

<Ntest> ::= ==

Produces equal.

<Ntest> ::= >

Produces greater_than.

<Ntest> ::= >=

Produces greater_than_or_equal.

3.2.12 Rounding_mode

<Rounding_mode> ::= ROUNDING_MODE?
(<Exp> , <Rounding_mode>, <Rounding_mode>)

<Rounding_mode> ::= <ntest_constructor> < (<constructor_param>-List) >-Opt

<Rounding_mode> ::= <ntest_token> < [<token_param>-List] >-Opt

There are no constructions for <Rounding_mode> other than the standard ones.

3.2.13 Shape

<Shape> ::= SHAPE ? (<Exp> , <Shape>, <Shape>)

<Shape> ::= <shape_constructor> < (<constructor_param>-List) >-Opt

<Shape> ::= <shape_token> < [<token_param>-List] >-Opt

The standard expansions.

<Shape> ::= Float

The shape for an IEEE 32 bit float.

<Shape> ::= Double

The shape for an IEEE 64 bit float.

<Shape> ::= <Sign>-Opt Int

<Sign> ::= Signed

<Sign> ::= Unsigned

The shape for a 32 bit signed or unsigned integer. The default is signed.

<Shape> ::= <Sign>-Opt Long

The shape for a 32 bit signed or unsigned integer.

<Shape> ::= <Sign>-Opt Short

The shape for a 16 bit signed or unsigned integer.

<Shape> ::= <Sign>-Opt Char

The shape for a 8 bit signed or unsigned integer.

<Shape> ::= Ptr <Shape>

The SHAPE pointer(alignment(<Shape>)).

3.2.14 Signed_Nat

<Signed_Nat> ::= SIGNED_NAT ? (<Exp> , <Signed_Nat>, <Signed_Nat>)

<Signed_Nat> ::= <signed_nat_constructor> < (<constructor_param>-List) >-Opt

<Signed_Nat> ::= <signed_nat_token> < [<token_param>-List] >-Opt

The standard expansions.

<Signed_Nat> ::= <integer_denotation>

<Signed_Nat> ::= - <integer_denotation>

This produces a make_signed_nat on the integer value.

<Signed_Nat> ::= <character>

<Signed_Nat> ::= - <character>

This produces a make_signed_nat on the ASCII value of the character.

<Signed_Nat> ::= LINE

This produces a make_signed_nat on the current line number of the file being compiled - useful for writing test programs.

3.2.15 Tag

<Tag> ::= <tag_token> < [<token_param>-List] >-Opt

The standard token application.

<Tag> ::= <ident>

This gives an obtain_tag; the <ident> must been declared as a TAG either globally or locally.

3.2.16 Token

TOKEN is rather a limited first-class sort. There is no explicit construction given for token_apply_token, since the only place where it can occur is in an expansion of a token parameter of another token; here it is produced implicitly. The only place where <Token> is expanded is in an actual TOKEN parameter of a token application; other uses (e.g. as in <shape_token>) are always <ident>s.

<Token> ::= <ident>

The <ident> must have been declarered by a <Tokdec> or <Tokdec> or is a formal parameter of TOKEN.

<Token> ::= Use <Tok_Defn>

This produces a use_tokdef. For <Tok_Defn> see section 3.1.2 on page 6. The critical use of this construction is to provide an actual TOKEN parameter to a token application where the <Tok_Defn> contains uses of tags or labels local to a procedure.

3.2.17 Transfer_mode

<Transfer_mode> ::= TRANSFER_MODE ? (<Exp> , <Transfer_mode>, <Transfer_mode>)

<Transfer_mode> ::= <transfer_mode_constructor> < (<constructor_param>-List) >-Opt

<Transfer_mode> ::= <transfer_mode_token> < [<token_param>-List] >-Opt

There are no expansions for <Transfer_mode> other than the standard expansions.

3.2.18 Variety

<Variety> ::= VARIETY ? (<Exp> , <Variety>, <Variety>)

<Variety> ::= <variety_constructor> < (<constructor_param>-List) >-Opt

<Variety> ::= <variety_token> < [<token_param>-List] >-Opt

The standard expansions.

<Variety> ::= <Signed_Nat> : <Signed_Nat>

This produces var_limits.

<Variety> ::= <Sign>-Opt Int

<Variety> ::= <Sign>-Opt Long

<Variety> ::= <Sign>-Opt Short

<Variety> ::= <Sign>-Opt Char

This produces the variety of the appropriate integer shape.

3.3 Control structure and local declarations

The control and declaration structure is given by <ClosedExp>:

<ClosedExp> ::= { <ExpSeq> }

<ExpSeq> ::= <Exp>-Opt
<ExpSeq> ::= <ExpSeq> ; <Exp>-Opt

This produces a TDF sequence if there is more than one <Exp>-Opt; if there is only one it is simply the production for <Exp>-Opt; any empty <Exp>-Opt produce make_top.

<ClosedExp> ::= <ConditionalExp>
<ClosedExp> ::= <RepeatExp>
<ClosedExp> ::= <LabelledExp>
<ClosedExp> ::= <Local_Defn>

The effect of these, together with the expansion of <Assertion> is given below.

3.3.1 ConditionalExp and Assertion

<ConditionalExp> ::= ? { <ExpSeq> | <LabelSetting>-Opt <ExpSeq> }

<LabelSetting> ::= : <Label> :

This produces a TDF conditional. The scope of a LABEL <ident> which may be introduced by <Label> is the first <ExpSeq>. A branch to the second half of the conditional will usually be made by the failure of an <Assertion> (ie a TDF _test) in the first half.

<Assertion> ::= <Query> (<Exp> <Ntest> <Exp> <FailDest>-Opt)

<Query> ::= ?

The assertion will be translated as an integer_test

<Query> ::= F?

The assertion will be translated as a floating_test with a wrap error_treatment.

<Query> ::= *?

The assertion will be translated as a pointer_test.

<Query> ::= .?

The assertion will be translated as an offset_test.

<Query> ::= P?

The assertion will be translated as a proc_test.

<FailDest> ::= | <Label>

The <Assertion> will produce the appropriate _test on its component <Exp>s. If the test fails, then control will pass to the <FailDest>-Opt. If <FailDest>-Opt is not empty, this is the <Label>. Otherwise, the <Assertion> must be in the immediate context of a <ConditionalExp> or <RepeatExp> with an empty <LabelSetting>-Opt; in which case this is treated as an anonymous label and control passes to there. For example, the following <Conditional> delivers the maximum of two integers:

?{ ?(a >= b); a | b }

This is equivalent to:

?{ ?(a >= b | L); a | :L: b }

without the hassle of having to invent the LABEL name, L.

3.3.2 RepeatExp

<RepeatExp> ::= Rep <Starter>-Opt { <LabelSetting>-Opt <ExpSeq> }

<Starter> = (<ExpSeq>)

This produces a TDF repeat. The loop will usually repeat by an <Assertion> failing to the <LabelSetting>-Opt; an empty <LabelSetting>-Opt will follow the same conventions as one in a <Conditional>. An empty <Starter>-Opt will produce make_top.

3.3.3 LabelledExp

<LabelledExp> ::= Labelled { <ExpSeq> <Places> }

<Places> ::= <Place>
<Places> ::= <Places> <Place>

<Place> ::= | : <Label> : <ExpSeq>

This produces a TDF labelled with the obvious parameters. The scope of any LABEL <idents> introduced by the <Label>s is the <LabelledExp>.

3.3.4 Local_Defn

A <Local_Defn> introduces an <ident> as a TAG for the scope of the component <ClosedExp>. Any containing an <Access> visible is also available globally - however it will only make sense in the constructor env_offset.

<Local_Defn> ::= Var <ident> <Access>-Opt <VarInit> <ClosedExp>

<VarInit> ::= = <Exp>

This <Local_Defn> produces a TDF variable with the obvious parameters.

<Local_Defn> ::= Var <ident> <Access>-Opt : <Shape> <VarInit>-Opt <ClosedExp>

Also a TDF variable. An empty <VarInit>-Opt gives make_value(<Shape>) as the initialisation to the variable. Using this form of variable definition also has the advantage of allowing one to use the simple form of the contents operation (* in section 3.2.7 on page 10).

<Local_Defn> ::= Let <ident> <Access>-Opt = <Exp> <ClosedExp>

This produces a TDF identify with the obvious parameters.

4 Example PL_TDF programs

4.1 Sieve of Erastothenes

```
/* Print out the primes less than 10000 */
String s1 = “%d\t”;                    /* good strings for printf */
String s2 = “\n”;

Var n: nof(10000, Char);               /* will contain1 for prime; 0 for composite */

Tokdef N = [ind:EXP]EXP n *+. (Sizeof(Char) .* ind);
                                       /* Token delivering pointer to element of n */

Iddec printf : proc;                   /* definition provided by ansi library */

Proc main = top ()
    Var i:Int
    Var j:Int
    { Rep (i = 2(Int))
            {          /* set i-th element of n to 1 */
             N[* i] = 1(Char);
             i = (* i + 1(Int));
             ?(* i >= 10000(Int))      /* NB assertion fails to continue loop */
            }
    Rep (i = 2(Int) )
            {
             ?{        ?( *(Char)N[* i] == 1(Char));
                       /* if its a prime ... */
                       Rep ( j = (* i + * i) )
                       { /*... wipe out composites */
                       N[* j] = 0(Char);
                       j = (* j + * i);
                       ?(* j >= 10000(Int))
             }
             | make_top
             };
             i = (* i + 1(Int));
             ?(* i >= 100(Int))
             };
```

```
Rep (i = 2(Int); j = 0(Int) )
        {       ?{      ?( *(Char)N[* i] == 1(Char));
                        /* if it's a prime, print it */
                        printf[top](s1, * i);
                        j = (* j + 1(Int));
                        ?{      ?( * j == 5(Int));
                                /* print new line */
                                printf[top](s2);
                                j = 0(Int)
                        | make_top
                        }
                | make_top
                };
                i = (* i + 1(Int));
                ?(* i >= 10000(Int))
        };
return(make_top)
};

Keep (main)         /* main will be an external name; so will printf since it is not defined */
```

4.2 Example with structures

```
Struct C (re:Double, im:Double);
        /* define TOKENs : C as a SHAPE for complex, with field offsets .re and .im
            and selectors re and im */

lddec printf:proc;

Proc addC = C (lv:C, rv:C)          /* add two complex numbers */
    Let l = * lv
    Let r = * rv
    { return( Cons[shape_offset(C)] ( .re: re[l] F+ re[r], .im: im[l] F+ im[r]) ) } ;

String s1 = "Ans = (%g, %g)\n";

Proc main = top()
    Let x = Cons[shape_offset(C)] (.re: 1.0(Double), .im:2.0(Double))
    Let y = Cons[shape_offset(C)] (.re: 3.0(Double), .im:4.0(Double))
    Let z = addC[C](x,y)
    {       printf[top](s1, re[z], im[z]);
                /* prints out "Ans = (4, 6)" */
            return(make_top)
    };

Keep(main)
```

4.3 Test for case

```
Iddec printf:proc;

String s1 = “%d is not in [%d,%d]\n”;
String s2 = “%d OK\n”;

Proc test = top(i:Int, l:Int, u:Int)          /* report whether l<=i<=u */
    ?{      ?(* i >= * l); ?(* i <= * u);
            printf[top](s2, * i);
            return(make_top)
    |       printf[top](s1, * i, * l, * u);
            return(make_top)
    };

String s3 = “ERROR with %d\n”;

Proc main = top()               /* check to see that case is working */
Var i:Int = 0(Int)
     Rep {
            Labelled {
                    Case * i (0 -> l0, 1 -> l1, 2:3 -> l2, 4:10000 -> l3)
                    | :l0: test[top](* i, 0(Int), 0(Int))
                    | :l1: test[top](* i, 1(Int), 1(Int))
                    | :l2: test[top](* i, 2(Int), 3(Int))
                    | :l3: printf[top](s3, * i)
            };
     i = (* i + 1(Int));
    ?(* i > 3(Int));
    return(make_top)
 };

Keep (main, test)
```

4.4 Example of use of high-order TOKENs

```
Tokdef IF = [ boolexp:TOKEN[LABEL]EXP, thenpt:EXP, elsept:EXP] EXP
                    ?{ boolexp[lab]; thenpt | :lab: elsept };
            /* IF is a TOKEN which can be used to mirror a standard if ... then ... else
                    construction; the boolexp is a formal TOKEN with a LABEL parameter
                    which is jumped to if the boolean is false */

Iddec printf: proc;

String cs = “Correct\n”;
String ws = “Wrong\n”;

Proc main = top()
    Var i:Int = 0(Int)
    {
            IF[ Use [l:LABEL]EXP ?(* i == 0(Int) | l), printf[top](cs), printf[top](ws) ];
                    /* in other words if (i==0) printf(“Correct”) else printf(“Wrong”) */
            IF[ Use [l:LABEL]EXP ?(* i != 0(Int) | l), printf[top](ws), printf[top](cs) ];
            i = IF[ Use [l:LABEL]EXP ?(* i != 0(Int) | l), 2(Int), 3(Int)];
            IF[ Use [l:LABEL]EXP ?(* i == 3(Int) | l), printf[top](cs), printf[top](ws) ];
            return(make_top)
     };

Keep (main)
```

4.5 A test for long jumps

```
Iddec printf:proc;

Proc f = bottom(env:pointer(frame_alignment), lab:pointer(code_alignment) )
{
     long_jump(* env, * lab)
};

String s1 = “Should not reach here\n”;
String s2 = “long-jump OK\n”;

Proc main = top()
Labelled{
                f[bottom](current_env, make_local_lv(l));
                printf[top](s1);              /* should never reach here */
                return(make_top)
           | :l:
                printf[top](s2);
                return(make_top)
           };

Keep (main)
```

5 Use of the PL_TDF compiler

Conventionally, PL_TDF programs are held in normal text files with suffix .pl. The PL_TDF compiler is invoked by:

```
pl [-v] [-Iinclude_path ...] [-g] infile.pl outfile.j
```

This compiles infile.pl to TDF in outfile.j. This file can be linked and loaded just as any other .j file using tcc.

The -v option will produce a cut-down pretty print of the TDF for the definitions and declarations of the tags, tokens and al_tags of the program on the standard output.

The -I options will defines the paths for any #include pre-processing directives in the text.

The -g option will put line number information into the TDF.

Compile-time error reporting is rather rudimentary and error recovery non-existent. Only the first error found will be reported on the standard error channel. This will give some indication of the type of error, together with the text line number and a print-out of the line, marking the place within the line where the error was detected.

Errors which can only be detected at translate-time are much more difficult to correct. These are usually shape or alignment errors, particularly in the construction of offsets. Try compiling and translating with the -g option. On the error, the translator will output the source filename and an approximate line-number corresponding to the position of the error in the PL_TDF.

Translating with the -g option may sometimes give warning messages from the system assembler being used; some assemblers object to being given line number information in anything else but the .text segment of the program. The main intention of the -g option is to detect and correct errors errors thrown up by the translators and not for run-time de-bugging, so do not regard a warning like this as a bug in the system.