Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A. Würtz, T.Murer

CHIPS Reference Manual

TIK-Report
Nr. 23, October 1996

Institut für Technische Informatik und Kommunikationsnetze
Computer Engineering and Networks Laboratory

A. Würtz, T. Murer
CHIPS Reference Manual
October 1996
TIK-Report Nr. 23

Computer Engineering and Networks Laboratory,
Swiss Federal Institute of Technology (ETH) Zurich

Institut für Technische Informatik und Kommunikationsnetze,
Eidgenössische Technische Hochschule Zürich

Gloriastrasse 35, ETH-Zentrum, CH-8092 Zürich, Switzerland

Table of Contents

Introduction

This document is intended to help you in using the CHIPS environment. It does not explain the purpose or background of CHIPS. If you are not familiar with that, please read the corresponding documents first. You can find them on our WWW server.

Lexical Definition: Scanners

A lexical definition contains descriptions of symbols and symbol classes that define all possible words of a language. The definition is written in CHIPSsL. A scanner can be generated directly from this definition. The different sections of a lexical definition will be explained using the following example:

```
SCANNER Calc;

  CHARACTERS
    digit = {'0'..'9'}.
    blankC = {020X}.

  IGNORE
    blankC

  TOKENS
    integer = digit {digit}.

  COMMENT FROM '(*' TO '*)' NESTED

  LITERALS
     '(', ')', '+', '-', '*', 'DIV', 'MOD'

END Calc.
```

The definition starts with the SCANNER keyword, followed by a name for the scanner. It is terminated by the END keyword, again followed by the name of the scanner.

In the CHARACTERS section, all valid characters are defined in the form of set definitions. Set definitions may contain characters and ranges of characters, separated by a comma. Characters may also be specified using their ASCII-code in hex format (see ‘blankC’ above). All subsequent sections may only use characters defined in this section.

```
digit = {'0'..'9'}.
hex = {'0'..'9', 'A'..'F'}.
blankC = {020X}.
```

The IGNORE section contains characters that should be ignored during syntax evaluation, i.e they are valid but are not returned as tokens or parts of tokens.

In the TOKENS section, you define tokens and classes of tokens using EBNF. You can use any set definition from the CHARACTERS section, repetition (‘{}’) and alternative (‘|’) signs.

```
integer = digit {digit}.
signed = ('+'|'-') digit {digit}.
real = digit {digit} '.' digit {digit}.
```

In the example above, the definition for the token class integer is not always sufficient. For example, a range of integers (e.g. ‘1..9’) would be erroneously recognised as the start of a real, since the first integer is followed by a dot. A sequence of digits followed by a dot is still an integer, if the dot is part of an ellipsis symbol ('..'). Such a situation can be described using the CONTEXT keyword:

```
integer = digit {digit} | digit {digit} CONTEXT ('..').
```

Token classes may also be Oberon Gadgets. Gadgets can be described by their NEW procedure or by an example of the gadget (i.e. placing a gadget directly into the scanner definition). There is also a keyword ('@gadget') to describe that just any gadget may be recognised.

```
        button = GADGET 'BasicGadgets.NewButton'.
        button = [picture goes here].
        any    = @gadget.
```

In the COMMENT section possible formats of comments are described. Comments must have an opening sign and may have a closing sign. If they don't have a closing sign, they continue until the end of the line or the end of the file. If they do have a closing sign, they may also be nested. Examples:

```
        COMMENT FROM '(*' TO '*)' NESTED.
        COMMENT FROM '//'.
```

The last section of a scanner definition is the LITERALS section. Here you define all tokens (literals) that do not belong to one of the previously defined token classes. You do not have to (and should not) define the TAB, EOL and EOF tokens, since they are predefined.

```
     LITERALS
         '(', ')', '+', '-', '*', 'DIV', 'MOD'
```

When the lexical definition is complete, an actual scanner implementation can be generated, which is then used as the base of a syntactic definition. Literals and tokens form the set of all *terminal symbols* in the syntactic definition. The description of a syntactic definition is described in the next section.

Syntactic Definition: Grammars

The context-free syntax of a language is described by a grammar. Grammars are specified in CHIPSpL. Again, the easiest way to describe the syntax of CHIPSpL is using an example. Below you see a grammar describing a simple calculating language that allows sentences like "1+1" or "2+4 = 3" etc.

```
GRAMMAR Calc;

  SCANNER Calc;

  SKIP EOL, TAB;

  NONTERMINAL CalcStart ::= Comparison.
  NONTERMINAL Comparison ::= Expression ['=' Expression].
  NONTERMINAL Expression ::= integer {('+'|'-') integer}.

END Calc.
```

The definition starts with the GRAMMAR keyword, followed by a name for the grammar. It is terminated by the END keyword, again followed by the name of the grammar.

The SCANNER statement specifies which scanner to use for the lexical analysis. In the example above, the scanner defined in the previous section is used. All symbols used within the syntactical definition must have been declared in the lexical definition.

The SKIP section list symbols that are to be skipped during syntax analysis.

The rest of the grammar consists of nonterminal definitions of the form

```
  NONTERMINAL <name> ::= <production>.
```

Notation

- Optional parts of a production are enclosed by '[' and ']' signs
- Alternatives are separated by a '|' sign
- Repetitions (zero or more) are enclosed by '{' and '}' signs.

Syntax definitions in CHIPSpL must have certain properties in order to be valid. These properties are verified during parser generation. Keep them in mind when designing your grammar.

Grammar Properties

1. **Correct Syntax**
   Of course, the first property that any specification must have is correct CHIPSpL syntax.
2. **Completeness**
   Every nonterminal that is used on the right side of a production must have a production of its own.
3. **Reachability**
   Every nonterminal defined in a production must be reachable (by following productions) from the start-nonterminal of the grammar.
4. **Terminalisability**
   All productions must be derivable to terminal symbols.

5. **Left-Recursion**
   Left-recursive declarations (e.g. A ::= A 'a'.) of nonterminal symbols are not allowed.
6. **LL(1)-Condition**
   For the explanation of this condition, two definitions are required which are provided here:
   The *First-Set* of a symbol sequence is the set of all terminal symbols that can be the start of the sequence or a sequence derived from that sequence.
   The *Follow-Set* of a symbol sequence is the set of terminal symbols that can follow that sequence.
   A grammar fulfills the LL(1)-condition, if
   a) for all of its productions the first-sets of all alternatives are mutually disjoint, and
   b) the follow-sets of nonterminal symbols that may be derived to the empty sequence are disjoint of the first-sets of all alternatives.

When the grammar definition is complete, a parser implementation can be generated, which is then used to parse the specified language and build a syntax tree.

Attributes and Evaluation Methods

In this section we describe how nonterminals can be assigned attributes and evaluation methods. These can be used to implement further language analysis and visualisation or transformation of contents into other languages. If you have read the correspondent information, you will have noticed that these methods are classified into transformation- and query-only. However, this classification is not implemented yet, all processes are treated equally.

Attributes

The syntax for attribute definition is as in the following example:

```
NONTERMINAL
  Expression::= integer {(‘+’|’-’) integer}.
ATTR value: INTEGER;
```

Attributes may have any valid Oberon 2 type. They are not preserved, i.e. not saved with the document.

Permanent Attributes

Permanent attributes are attributes that are saved with the document. Since the syntax tree is destroyed whenever a document is closed, permanent attributes cannot be stored in the nodes of the syntax tree. Instead they reside within container gadgets in the document text, and every nonterminal has to look for and reassign its permanent attribute container after each creation of the syntax tree. Attribute containers have a visible representation in the Oberon text, which depends on their usage state. An unused container is represented by a red circle, a container in use shows as a red bullet (filled circle).

Only basic Oberon types may be used for permanent attributes. Routines for finding an manipulating attribute containers can be found in the module `CHIPSAttributes` and are described below.

Finding an Unused Permanent Attribute Container

After creation, every nonterminal node in the syntax tree receives a message of type `Structures.AttrMsg` (see also below "Handler Concept"). If it has permanent attributes, it should call `FindAttrMark` with that message as first parameter (`M`). The second parameter (`syn`) is the node itself. The third parameter determines if an attributes container should be created if none is found. Set this to `FALSE` if your attributes are optional, set it to `TRUE` otherwise.

```
  PROCEDURE FindAttrMark(VAR M: Structures.AttrMsg;
    syn: CHIPS.NONTERM; insert: BOOLEAN);
```

Retrieving Permanent Attribute Values

You cannot access a nonterminal's permanent attributes directly. You have to use one of the following routines for retrieving their value, depending on the type of the attribute. The nonterminal node should be passed as the first parameter (`obj`). `Name` should contain the name of the desired attribute value. If the routine returns `TRUE`, `res` contains value of the attribute specified by `name`. If it returns `FALSE`, the attribute container was not found, did not contain an attribute with that name or the attribute had the wrong type.

```
  PROCEDURE GetStringAttr(obj: CHIPS.NONTERM; name: ARRAY OF CHAR;
    VAR res: ARRAY OF CHAR): BOOLEAN;
  PROCEDURE GetIntAttr(obj: CHIPS.NONTERM; name: ARRAY OF CHAR;
    VAR res: LONGINT): BOOLEAN;
  PROCEDURE GetRealAttr(obj: CHIPS.NONTERM; name: ARRAY OF CHAR;
    VAR res: LONGREAL): BOOLEAN;
  PROCEDURE GetBoolAttr(obj: CHIPS.NONTERM; name: ARRAY OF CHAR;
    VAR res: BOOLEAN): BOOLEAN;
  PROCEDURE GetCharAttr(obj: CHIPS.NONTERM; name: ARRAY OF CHAR;
    VAR res: CHAR): BOOLEAN;
```

Setting Permanent Attribute Values

Again, you cannot access permanent attributes directly. You have to use one of the following routines for setting their value, depending on the type of the attribute. The nonterminal node should be passed as the first parameter (`obj`). `Name` should contain the name of the desired attribute value, `val` should contain the desired value for the attribute specified by `name`. If the routine returns `FALSE`, the attribute container was not found or an attribute with that name already existed but had the wrong type.

```
  PROCEDURE SetStringAttr(obj: CHIPS.NONTERM;
    name, val: ARRAY OF CHAR): BOOLEAN;
  PROCEDURE SetIntAttr(obj: CHIPS.NONTERM; name: ARRAY OF CHAR;
    val: LONGINT): BOOLEAN;
  PROCEDURE SetRealAttr(obj: CHIPS.NONTERM; name: ARRAY OF CHAR;
    val: LONGREAL): BOOLEAN;
  PROCEDURE SetBoolAttr(obj: CHIPS.NONTERM; name: ARRAY OF CHAR;
    val: BOOLEAN): BOOLEAN;
  PROCEDURE SetCharAttr(obj: CHIPS.NONTERM; name: ARRAY OF CHAR;
    val: CHAR): BOOLEAN;
```

Attribute Evaluation Methods

Of course, if a nonterminal has attributes, it needs a means of evaluating their value. This is done by attribute evaluation methods. The definition of those methods is described in this section. This mechanism is used for all kinds of semantic behaviour you want to define for a nonterminal, be it attribute evaluation or query.

As you have read, every nonterminal description is treated as a class definition, every nonterminal in an actual syntax tree is an instance of that class. Attribute evaluation methods are actually methods of that class. However, the implementation in CHIPS is somewhat different from that in Oberon 2 and has some restrictions. Again, let's start with an example: below you see an evaluation method for the attribute `value` of the nonterminal `Expression`:

```
NONTERMINAL
  Expression::= integer {('+'|'-') integer}.
ATTR value: INTEGER;

PROCEDURE EvalValue(self: Expression; env: CHIPS.ENVIRONMENT);
VAR
  number, op: CHIPS.TERM;
  str: CHIPS.STRING;
  sum: INTEGER;
BEGIN
  number:=CHIPS.FIRSTTERM(env, self);
  CHIPS.GETSTRING(env, number, str);
  sum:=StringToInt(str);
  op:=CHIPS.NEXTTERM(env, self, number);
  WHILE op#NIL DO
    number:=CHIPS.NEXTTERM(env, self, op);
    CHIPS.GETSTRING(env, number, str);
    IF CHIPS.ISTERM(op, '+') THEN
      sum:=sum+StringToInt(str)
    ELSE
      sum:=sum-StringToInt(str)
    END;
    op:=CHIPS.NEXTTERM(env, self, number)
  END;
  self.value:=sum
END EvalValue;
```

First let us look at what the above routine does: it goes through each terminal of its production, which consists of at least one integer, followed by zero or more pairs of an operand and an integer. For each integer it calculates the corresponding value using `StringToInt` (not specified here) and adds/subtracts it to/from the local variable `sum`, depending on the type of the operand. The final value of sum is stored into the instance's `value` field. Thus this method evaluates the attribute `value` of the nonterminal `Expression`. It uses several routines from the CHIPS module for walking the syntax tree and retrieving information about nodes in the tree. The following section describes all routines and types exported by that module.

Parameters and Result types

There are no restrictions (apart from the usual Oberon ones) for a nonterminals methods, as far as parameters and result types are concerned. However, you must pass a reference to the nonterminal object itself in order to be able to access its instance fields. It is also recommendable to pass a reference to the current environment, because it is often required by routines exported from the CHIPS module. More about environments and the CHIPS module will follow below.

The Environment

A special class called Environment exists which serves as a gateway between the CHIPS environment and an instance of your grammar definition (i.e. a text document written in the editor that was generated from your grammar definition). To the CHIPS environment it serves as an entry point to your grammar class: it initiates creation of the syntax tree when necessary and passes requests from the CHIPS environment on to the start nonterminal. Your grammar class needs a reference to this Environment for calls to most of the routines in CHIPS.Mod.

Extending an existing Grammar

If you are extending an existing Grammar, the specification syntax is somewhat different. Imagine, we would add the attribution method `EvalValue` in an extension `NewCalc` of `Calc` instead of `Calc` itself:

```
GRAMMAR NewCalc(CalcStrucs);

   NONTERMINAL Expression(CalcStrucs.Expression)

   ATTR value: INTEGER;

   PROCEDURE EvalValue(self: Expression; env: CHIPS.ENVIRONMENT);
   VAR
      (* ... *)
   BEGIN
      (* ... *)
   END EvalValue;

END NewCalc.
```

The base grammar for NewCalc is indicated by the name in brackets behind the grammar identifier. The base class for the nonterminal Expression is indicated by the qualified name in brackets behind the nonterminal identifier. *NOTE:* You have to append 'Strucs' to every occurrence of the base grammar's name. Instead of 'Calc' you have to write 'CalcStrucs'.

Exporting fields an methods

Export control is done as usual in Oberon: to export a method or attribute, put an asterisk (*) behind its name in the definition. To export an attribute for read-only access, put a dash (-) behind its name in the attribute definition.

Overriding Methods

To override an existing method, redefine it with the same name and parameters, but put an asterisk in front of its name in the definition.

Calling methods

This is why you must always pass 'self' as a parameter to your object methods: to call another method, write 'self.methodname'.

Calling inherited methods

To call the inherited version from within an override, write the base class' name, append 'Strucs', write the method name and append the nonterminal's name. Example: if you extend `NewCalc` and override `Expression.EvalValue`, you call the inherited method with `NewCalcStrucs.EvalValueExpression`.

The Handler Interface

Before we come to the handler interface, we must explain how attribution is invoked. When the user opens a text document based on your grammar, he/she finds a menu called "Known Messages" at the top edge of the document window with at least one entry for parsing the text. All further entries refer to attribution methods you implement in your grammar. The start nonterminal of your grammar is responsible for providing those entries, for activating your attribution processes when the user chooses one of your entries. Additionally, there is the possibility for specifying a precondition for each attribution process, i.e. another attribution process that has to be performed successfully before.

For every attribution process you must declare and export a special message record that extends `CHIPS.OBJMSG`. The best way is to do this in a special module, for example `CalcMsgs.Mod`.

```
MODULE CalcMsgs;

  IMPORT
    DistMessages, CHIPS;

  TYPE
    EvalValueMsgPtr* = POINTER TO EvalValueMsg;
    EvalValueMsg* = RECORD (CHIPS.OBJMSG)
    END;

  PROCEDURE NewEvalValueMsg*;
  VAR
    M: EvalValueMsgPtr;
  BEGIN
    NEW(M);
    ContextProt.InitObjMsg(M);
    DistMessages.New:=M;
  END NewEvalValueMsg;

END CalcMsgs.
```

In the example above, the message record was called `EvalValueMsg`. In addition, you must declare and export a pointer to that message, called <msgname>`Ptr`, in our example `EvalValueMsgPtr`. And you must declare and export a New-procedure for your message, called `New`<messagename>, that does at least what's shown above. If you have additional fields in your message (for providing parameters or receiving results of the attribution process), this is also a good place to initialise them to a decent value. The name of the message record and module is also the name of the user command that appears in the 'Known Messages' Menu. So the command for invoking the EvalValue process is 'CalcMsgs.EvalValueMsg'. Keep this in mind for the next section.

The start nonterminal of your grammar has method called handle, which is called with different types of messages to retrieve the above information or to invoke parsing or attribution processes. If you want to provide a user interface for own attribution methods, you must override this handle method. Again, an example (showing the handle method of the `Calc` grammar's start nonterminal `CalcStart`) is used to explain the handling of the different message types:

```
PROCEDURE *handle*(SELF: CHIPS.NONTERM; VAR M: Objects.ObjMsg);
BEGIN
  WITH SELF: CalcStart DO
    IF M IS CHIPS.OBJMSG THEN
      IF M IS CalcMsgs.EvalValueMsg THEN
        WITH M: CalcMsgs.EvalValueMsg DO
          SELF(CalcStart).EvalValue(SELF(CalcStart), M.env);
          M.res:=0
        END
      ELSE
        CHIPS.HANDLENONTERM(SELF, M)
      END
    ELSIF M IS CHIPS.ENUMMSG THEN
      WITH M: CHIPS.ENUMMSG DO
        CHIPS.HANDLENONTERM(SELF, M);
        M.Enum(M.obj, 'CalcMsgs.EvalValueMsg')
      END
    ELSIF M IS CHIPS.REQUIRESMSG THEN
      WITH M: CHIPS.REQUIRESMSG DO
        IF M.cmd = 'CalcMsgs.EvalValueMsg' THEN
          COPY(CHIPS.CHECKSYNTAX, M.cmd); M.res:=0
        ELSE
          CHIPS.HANDLENONTERM(SELF, M)
        END
      END
    ELSE
      CHIPS.HANDLENONTERM(SELF, M)
    END
  END
END handle;
```

CHIPS.OBJMSG
Whenever the start nonterminal receives a `CHIPS.OBJMSG` it knows that an attribution process is to be invoked. Every `OBJMSG` carries a reference to the document's environment (`env` field), which you may use when calling any of the routines in the CHIPS module. The exact type of the message determines what attribution is to be performed. `CalcStart` knows `CalcMsgs.EvalValueMsg`, so it calls `SELF.EvalValue` and sets the `res` field of the message to zero (message handled, no error). If an error occurred, it would set the field to a positive value. For all other subtypes of `OBJMSG`, the inherited handler is called, because `CalcStart` does not know them.

CHIPS.ENUMMSG
The CHIPS environment calls the handler with this message type to find out, what types of messages the start nonterminal understands. The start nonterminal first passes the message to the inherited handler and then calls the `Enum` routine of the message with every message name it understands.

CHIPS.REQUIRESMSG
The CHIPS environment calls the handler with this type of message to find out about preconditions for a certain attribution process, i.e. what other attribution process must have been performed successfully before invoking this one. On entry the `cmd` field contains the name of the message to invoke the attribution. On exit, it must contain the name of the message to invoke the attribution that is the precondition. If an attribution does not have any precondition, copy CHIPS.CHECKSYNTAX to the `cmd` field. Don't forget to set the res field to zero, otherwise CHIPS thinks you do not know the message name that was passed in.

The CHIPS Programming Interface

As mentioned above, certain routines are available to ease the implementation of attribution methods. Most of these routines provide access to the syntax tree that is built during parsing. All routines are exported by the module CHIPS.Mod. Below you find a complete list with descriptions.

Constants

```
  CONST CHECKSYNTAX
```

The standard command that invokes a syntax check with the current environment. Return this string in a REQUIRESMSG if your attribution method has no preconditions.

Types

```
  TYPE ENVIRONMENT
```

Object type of an environment. Most of the routines below exspect a parameter of this type. You receive a reference to your environment with every Message you get.

```
  TYPE SYMBOL
```

Base type for all symbols (Terminals and Nonterminals).

```
  TYPE NONTERM
```

Base type for all Nonterminals. Every Nonterminal you define is derived from this type.

```
  TYPE TERM
```

Base type for all Terminals.

```
  TYPE OBJMSG
         env: ENVIRONMENT;
         res: INTEGER;
       END;
```

Base type for all messages that you define for invoking your attribution processes. The env field contains a reference to your environment. The res field should be set to zero after successful completion or to a positive number if an error occurred. Do not touch this field if an unknown attribution process is demanded.

```
  TYPE ENUMMSG
         Enum: PROCEDURE(obj: Object.Object; cmd: ARRAY OF CHAR);
         obj: Objects.Object;
       END;
```

Message used to enumerate all known Message types for an environment. Call the Enum procedure once for every message type you know. Pass the message's obj field as first parameter and the message type (as string) as second parameter.

```
  TYPE REQUIRESMSG
         cmd: ARRAY 64 OF CHAR;
         res: INTEGER;
       END;
```

Message used to find out about the precondition for a certain attribution method. The cmd field contains the message type for the method in question. Upon return, it should contain the attribution method that must have been executed successfully **immediately** before this one. If your method has no preconditions, pass CHECKSYNTAX in the cmd field. Set the res field to zero to indicate that you know the method at all.

Handlers

```
   VAR HANDLENONTERM-: PROCEDURE (N: NONTERM; VAR M: Objects.ObjMsg);
```

Handle method for the standard nonterminal. Call this method as 'inherited' from your nonterminal's handle procedure (as shown in the handle implementation example above).

Error Handling

```
   PROCEDURE ERROR (node: NONTERM; message: ARRAY OF CHAR);
```

Error indication procedure. This procedure converts the message into an error number and shows a corresponding error item in the text, just behind the nonterminal you pass in as node. When the user clicks in that item, he/she gets the error message in the log window.

Terminal Handling

```
   PROCEDURE FIRSTTERM (E: ENVIRONMENT; N: NONTERM): TERM;
   PROCEDURE PREVTERM (E: ENVIRONMENT; N: NONTERM; sym: SYMBOL): TERM;
   PROCEDURE NEXTTERM (E: ENVIRONMENT; N: NONTERM; sym: SYMBOL): TERM;
   PROCEDURE LASTTERM (E: ENVIRONMENT; N: NONTERM): Tokens.Token;
```

The four routines above are used to navigate within terminals of a nonterminal's production. They return only terminals that are mentioned directly in the production and do not belong to another nonterminal of the production. Pass your environment in E and the nonterminal whose production is concerned in N. Sym indicates the start position for the search for the next/previous terminal and may be either a terminal or a nonterminal. If the start or end of the production is reached, or the production does not contain any terminals, NIL is returned.

```
   PROCEDURE ISTERM (t: TERM; Type: ARRAY OF CHAR): BOOLEAN;
```

Provides type test for terminals. Pass the terminal in question in t, the desired type in Type. If the type is a literal, pass in the literal as a string. However, if the type is a token class, pass the name of the token class, not enclosed by quotes.

Nonterminal Handling

```
   PROCEDURE FIRSTNT (N: NONTERM): NONTERM;
   PROCEDURE PREVNT (N: NONTERM; sym: SYMBOL): NONTERM;
   PROCEDURE NEXTNT (N: NONTERM; sym: SYMBOL): NONTERM;
   PROCEDURE LASTNT (N: NONTERM): NONTERM;
```

The four routines above are used to navigate within nonterminals of a nonterminal's production. They return only nonterminals that are mentioned directly in the production and do not belong to another nonterminal of the production. Pass the nonterminal whose production is concerned in N. Sym indicates the start position for the search for the next/previous nonterminal and maybe either a terminal or a nonterminal. If the start or end of the production is reached, or the production does not contain any nonterminals, NIL is returned.

Note: a nonterminal that is not optional in your production, but can be reduced to an empty nonterminal in it's own production, has to be treated as if it was optional.

Type checking: use the normal Oberon IS statement to test for type membership.

SKIP Symbol Handling

```
   PROCEDURE FIRSTSKIP (E: ENVIRONMENT; N: NONTERM): TERM;
   PROCEDURE PREVSKIP (E: ENVIRONMENT; N: NONTERM; sym: SYMBOL): TERM;
   PROCEDURE NEXTSKIP (E: ENVIRONMENT; N: NONTERM; sym: SYMBOL): TERM;
   PROCEDURE LASTSKIP (E: ENVIRONMENT; N: NONTERM): Tokens.Token;
```

For some types of attribution methods (e.g. formatting tools) it may be necessary to have information also about those terminals that do not contribute to the production and are normally skipped (mostly TABs or EOLs or EOFs). The above routines are used to find these terminals. They return only skip symbols that are found directly in the production and do not belong to another nonterminal of the production. Pass your environment in E and the nonterminal whose production is concerned in N. Sym indicates the start position for the search for the next/previous skip symbol and may be either a terminal or a nonterminal or a skip symbol. If the start or end of the production is reached, or the production does not contain any skip symbols, NIL is returned. Please note that <space> is normally not a skip symbol but is already ignored by the scanner.

Text Handling

```
  PROCEDURE GETSTRING (E: ENVIRONMENT; node: TERM; VAR s: ARRAY OF CHAR);
```
Returns the string representation of a terminal. Only works for terminals and skip symbols. For nonterminals, you must use GETTEXT in conjunction with RANGE to acquire the textual representation.

```
  PROCEDURE GETTEXT (E: ENVIRONMENT): Texts.Text;
```
Returns a reference to the textual representation of a document.

```
  PROCEDURE RANGE (node: SYMBOL; VAR from, to: LONGINT);
```
Returns the start and end location of a terminal or nonterminal within the textual representation of a document.

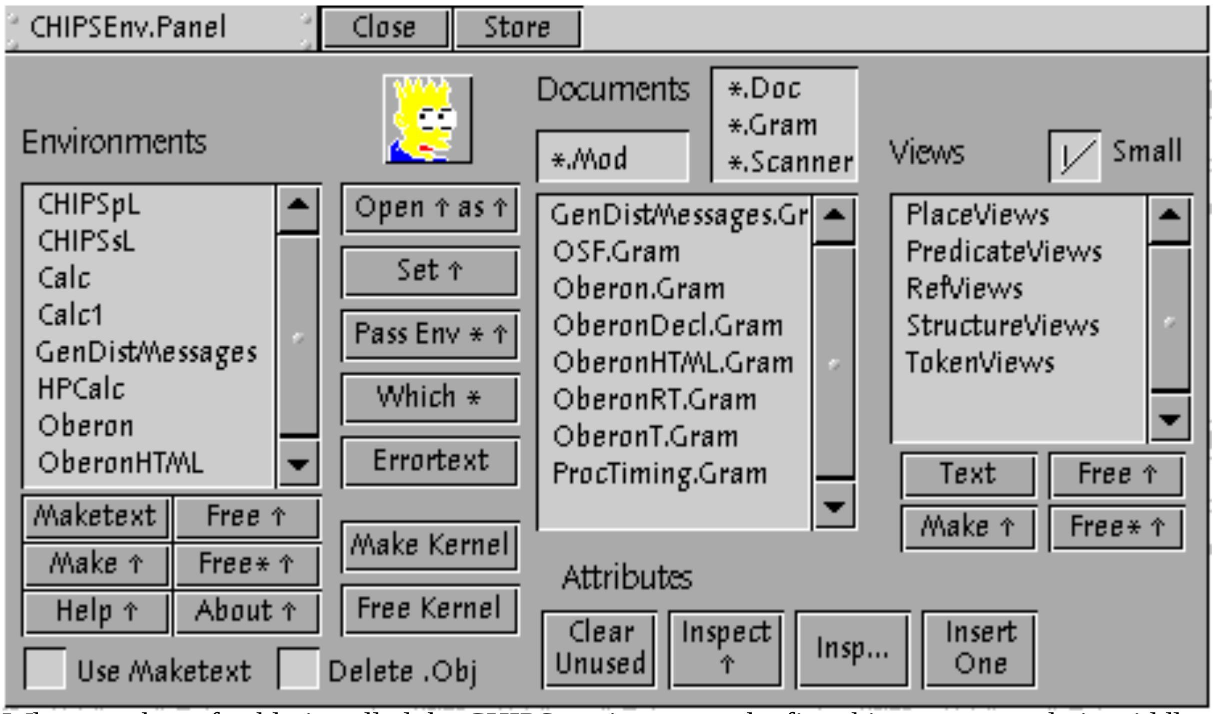
```
  PROCEDURE PRESPACE (node: SYMBOL; VAR from, to: LONGINT);
```
Returns the start and end location of white space before a terminal or nonterminal within the textual representation of a document. White space means characters that have been ignored by the scanner..

```
  PROCEDURE RELPOS (anchor, node: SYMBOL): INTEGER;
  CONST Undef = 0;
  CONST Before = 1;
  CONST AtStartOf = 2;
  CONST Inside = 3;
  CONST AtEndOf = 4;
  CONST Behind = 5;
  CONST Same = 6;
  CONST Enclosing = 7;
```
Returns the relative position of `node` with respect to `anchor`. If `anchor` is a terminal, `node` may be `Before`, `Same` or `Behind` `anchor` or - if `node` is a nonterminal - `Enclosing` `anchor`. If `anchor` is a nonterminal, `node` may also be `AtStartOf`, `Inside` or `AtEndof anchor`. If an error occurs, `Undef` is returned.

The CHIPS Environment

In this section we discuss the CHIPS environment as it appears to the user. Main user interface component is the CHIPSEnv Panel which is shown below.

The CHIPSEnv Panel



When you have freshly installed the CHIPS environment, the first thing you must do is middle-click on Bart Simpson's head. This compiles the essential CHIPS modules that are needed to provide further panel functionality. The Panel has four sections whose meaning is described below.

Environments Section

This section has all the functionality to build environments and choose environments for a new document. Right-click the environment you wish to work with in the environments list. If you define an environment of your own, you can easily add it using the standard Oberon move procedure (write down the name of your grammar, select the text and do a left-click with a middle-interclick on the Environments list)

Maketext

Shows the contents of the file <environment>.Make. This file contains the names of the modules that must be compiled before the actual environment is compiled. If you have defined attribution messages for your grammar, the name of the module containing those message definitions (usually <environment<Msgs.Mod) should be in the file.

Make

Actually builds the environment, i.e. compiles all the necessary modules. The setting of the option *Use Maketext* determines if also the modules contained in the Maketext are compiled.

Help
Has no function at the moment.

Free
Frees the selected environment (Scanner and Parser)

*Free**
Frees the selected environment and all its imported modules.

About
Has no function at the moment.

Use Maketext
Check this item if the modules contained in the Maketext should also be compiled during Make.

Delete .Obj
Check this item if the previous object files for the environment should be deleted before Make.

Documents Section
In the documents section you find a file list. Middle-click on an item in the list to open the corresponding document.

Fixed Filterlist
Here you find the three main document extensions used by CHIPS: .Gram is used for documents containing a grammar description, .Scanner is used for documents containing a scanner definition and .Doc is used for documents opened in an environment that you specified yourself. Middle-click on one of the filters to update the file list accordingly.

Variable Filterlist
Similar to the fixed list except that you may enter any file filter you like. Press return to update the file list according to your new file filter. Note: document opened by middle-clicking the file list are always opened as CHIPS documents, regardless of their document type. It is thus useless to open an Oberon System 3 panel from this list.

Open ^ as ^
This button refers to both, the Environments and the Documents list. Select an environment and a document and middle-click on this button to open the document with the selected environment. You only have to use this button if you want to open the document with an environment that is different from the one that is was created in. Other wise just middle-click on the document itself.

Set ^
Use this button to change the environment for an open document. The star marker (*) must be set on the document (use F1 to set it at the mouse location) and an environment must be selected in the Environments list.

*Pass Env * ^*
Use this button to create an environment and attach it to an Oberon System 3 panel. This is used for views only and is thought as a debugging help. The star marker must be set on the panel (use F1 to set it at the mouse location) and an environment must be selected in the Environments list.

*Which **
Use this button to find out about the environment behind a CHIPS document or panel. The star marker must be set on the document or panel (use F1 to set it at the mouse location).

ErrorText
Open the error text for the selected environment. The error text is generated during scanner generation and contains all error numbers with their corresponding strings.

Views Section
In addition to the textual representation, CHIPS provides four different standard views for a document. These views are listed in the Views list and described in the CHIPS Views section. If you define views of your own, you can easily add them using the standard Oberon move procedure (write the name of your view, select the text and do a left-click with a middle-interclick on the views list).

You open a view for a document by placing the star marker on the document's current representation and middle-clicking on the view in the list.

Small
Check this item if you are working with a rather small screen (15" or less). Views will be opened smaller and their contents will appear smaller.

Text
Opens a textual representation for the marked document.

Free ^
Frees the view implementation module.

Free ^*
Frees the view implementation module and all modules that it imports.

Attributes Section
The Attributes sections provides functionality for manipulating attributes of nonterminals. As you know, permanent attributes are represented by attribute markers, little red circles in the text.

Clear unused
Clears all unused attribute containers in the text that has the star marker (use F1 to set the star marker at the mouse location). WARNING: never use this button on a text that has not been parsed recently! If text hasn't been parsed, has been changed after parsing or an error occurred, all attribute markers are unused and will thus be deleted.

Inspect ^
This button is for inspecting non-permanent attributes of a nonterminal. Non-permanent attributes cannot be changed directly. Select an arbitrary portion of text in a CHIPS document using the right mouse button. Clicking on the Inspect button will find the largest nonterminal that fits into the selection and display its attributes in the Oberon log window.

Insp...
This button is for inspecting the permanent attributes of a nonterminal. Select an attribute container by right-clicking on it. Middle-clicking on the Insp… button reveals on Oberon

Inspector. Middle-click on the inspector's Inspect Model button to see and set the permanent attributes contained in the container.

Insert One
Inserts a new attribute container at the cursor location in the current document. Use this only for nonterminals that have only optional permanent attributes. Containers for non-optional attributes are generated automatically during parse.

Invoking Attribution Methods

Each CHIPS Document has a pull-down menu called 'Known Messages' in its title bar. This menu shows all available attribution processes. To invoke such a process, choose the corresponding item in this menu and watch the Oberon log window.

Generating a Scanner

From within the scanner definition (e.g. My.Scanner), choose 'CHIPSsLMsgs.GenScannerMsg' to create the Oberon implementation of your scanner. It is contained in a file named 'MyScanner.Mod'.

Generating a Parser

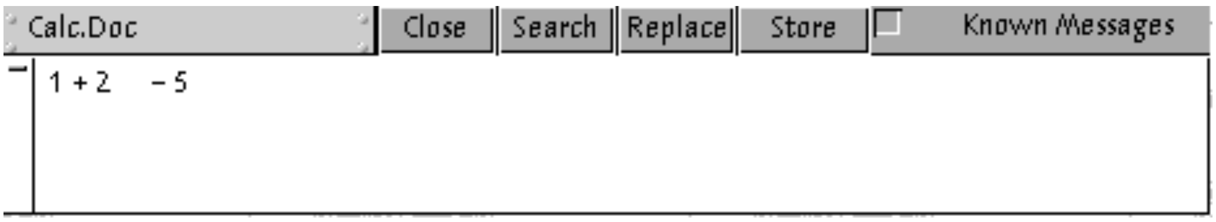
From within the grammar definition (e.g. My.Gram), choose 'CHIPSpLMsgs.GenParserMsg' to create the Oberon implementation of your parser and 'CHIPSpLMsgs.GenErrorsMsg' to create the corresponding error text. The parser implementation is contained in a file named 'MyStrucs.Mod', the error text is contained in a file named 'My.Errors'.

CHIPS Views

The CHIPS kernel provides three standard views for all CHIPS documents and one specially for CHIPS grammars. In addition you may define your own views. In this section, the standard views and the procedure for defining an own view is described.

Standard Views

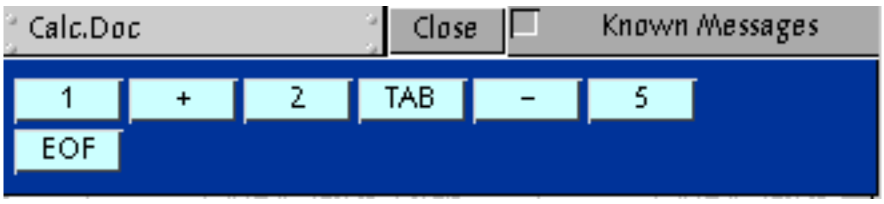
The standard views provide a means to look at the internal structures that are built from the document's textual representation as well as for checking if the list of preconditions has been built correctly. Again, we use the Calc example for explaining the function of those views.



```
1 + 2	- 5
```

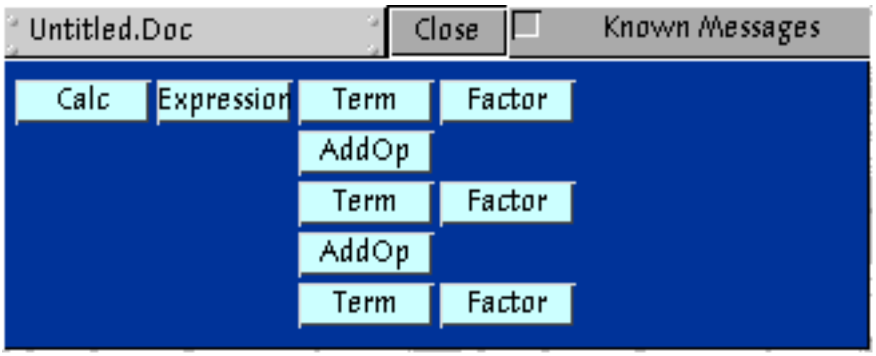
TokenView

The TokenView shows the document contents as a list of tokens as they are provided by the scanner. A line of tokens corresponds to a line of text. If you middle-click on one of the items in the view, the corresponding text is selected in the textual view of the document.

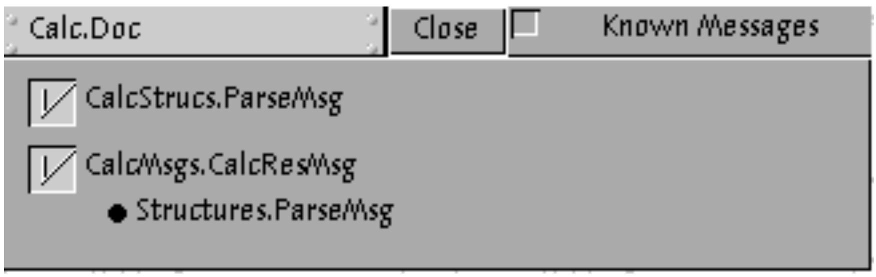


StructureView

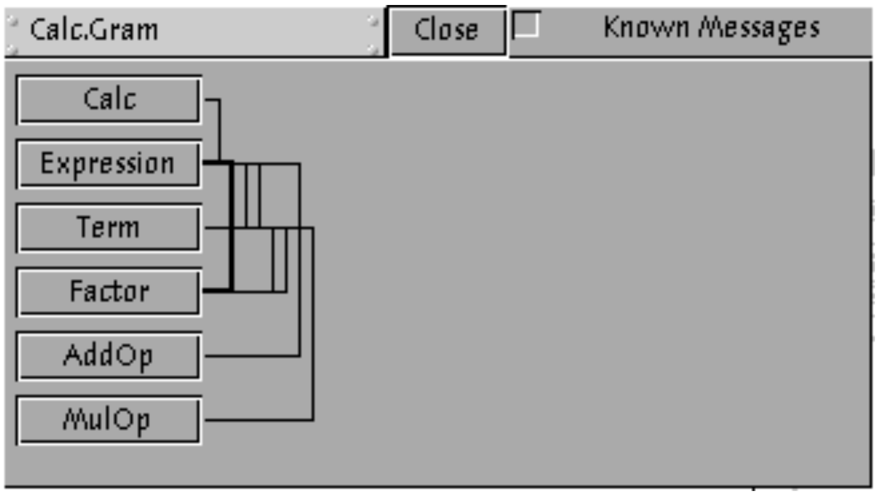
The structure (or syntax) tree for the selected document is shown with all nonterminal nodes. The tree lies on its side, i.e. the left-most column shows the production of the start-nonterminal, the second column shows it sub productions and so on. As with the TokenView, if you middle-click on one of the items in the view, the corresponding text is selected in the textual view of the document.



PredicateView
The PredicateView provides an overview of the precondition hierarchy that has been built from your environment's responses to the CHIPS.ENUMMSG and CHIPS.REQUIRESMSG calls. Checkboxes show attribution methods and their corresponding bulleted list shows their preconditions. Checked boxes/black bullets indicate successful completion of the corresponding attribution method.



RefView
This view can only be applied to CHIPSpL documents and shows, where nonterminals occur in the production of other nonterminals.



Programming Your own Views

The module CHIPSViews implements the more common tasks of a CHIPS view, such as:

- Notification of commands
- keeping the representation up-to-date

The main task in implementing an own view is overriding the view's ShowView method to visualise the syntax tree in the desired form. However, the view has no direct access to the syntax tree. It has only a reference to the document's text in its own obj field. The text is an intelligent object, that is capable to pass messages along to the document's start nonterminal. Thus, all information retrieval from the document's syntax tree must be implemented as a query attribution method of the start nonterminal with a corresponding query message. The view panel can then send that query message to the text to get all the information it needs to visualise certain document attributes.

```
DEFINITION CHIPSViews;

  TYPE
    Panel = POINTER TO PanelDesc;
    PanelDesc = RECORD(Panels.PanelDesc)
      ShowCmd: ARRAY 32 OF CHAR;
      ShowView: PROCEDURE (P: Panel)
    END;

  PROCEDURE InitPanel(P: Panel);
  PROCEDURE NewPanel;
  PROCEDURE PanelHandler(obj: Objects.Object; VAR M: Objects.ObjMsg);
  PROCEDURE RemoveFrames(P: Panel);
  PROCEDURE Show;  (* ^ CHIPSViews.PanelNewProc DocName *)
  PROCEDURE ShowDoc(P: Panel);

END CHIPSViews.
```

The field ShowCmd of the view panel contains a precondition, i.e an attribute method that must be successfully performed before the view can show its contents. If that precondition is not true, the view remains empty. The precondition should not the query method the view uses, but that query method's precondition. Again, if your query method does not have any precondition, fill CHIPS.CHECKSYNTAX into the ShowCmd field. The ShowView field does the actual work. It's the view's display method and must be overridden.

A Skeleton for an own View

This section is meant to provide an overview of tasks to be done when implementing an own CHIPS view. Just look at the source and read the comments.

```
MODULE ViewSkeletton;

   IMPORT
      CHIPSViews, CHIPS, Objects, Gadgets, MyMsgs;

   PROCEDURE ShowView(SELF: CHIPSViews.Panel);
   VAR
      L: Objects.LinkMsg;
      M: MyMsgs.QueryMsg;
   BEGIN
      (* get our obj field *)
      L.name:='Model'; L.id:=Objects.get; L.res:=-1;
      SELF.handle(P, L);
      IF (L.res=0) & (L.obj#NIL) THEN
         (* query information needed for view *);
         M.res:=-1; (* not handled *)
         M.resultList:=NIL; (* e.g. list of attributes we need *)
         L.obj.handle(L.obj, M);
         IF M.res=0 THEN
            (* draw view using information we got in resultList *)
         END
      END
   END ShowView;

   PROCEDURE InitPanel*(P: CHIPSViews.Panel);
   (*init pabel fields for a ViewSkeletton panel *)
   BEGIN
      IF P#NIL THEN
         Panels.Freeze(P,TRUE); (* no user manipulation allowed *)
         P.ShowCmd:='MyMsgs.PreCondMsg';  (* or CHIPS.CHECKSYNTAX *)
         P.ShowView:=ShowView; (* set the draw method *)
      END
   END InitPanel;

   PROCEDURE NewPanel*;
   (* create a CHIPS panel and init it to be a ViewSkeletton panel *)
   VAR
      obj: Objects.Object;
   BEGIN
      obj:=Gadgets.CreateObject('CHIPSViews.NewPanel');
      IF obj#NIL THEN
         InitPanel(obj(CHIPSViews.Panel))
      END
   END NewPanel;

END SkeletonViews.
```