HLA v2.0 Grammar

This document describes the grammar for the HLA v2.0 language. This documentation assumes that the reader is familiar with the concept of regular languages, context-free languages, context-free grammars, regular expressions, EBNF, and similar concepts from automata and compiler theory.

Context-Free Grammar Notation for the HLA v2.0 Grammar

This documentation uses a modified notation for stating context free grammars. Since this notation is somewhat unique to HLA v2.0, it is important to describe the notation in use prior to presenting the actual grammar.

A context-free grammar (CFG) named G is traditional defined as the following four-tuple:

```
G = {N, T, S, P}
```

where N is a set of non-terminal symbols, T is a set of terminal symbols, S is a starting symbol (a non-terminal), and P is a set of productions. This document assumes that the reader is familiar with these terms.

This document will not provide a formal list of the non-terminal or terminal symbols in the HLA grammar. Instead, this document will use typographical notation to differentiate the two (and differentiate them from other, meta-symbols, in the grammar).

Non-terminal symbols will always be identifiers appearing in a normal (i.e., not bold) font. An identifier is a non-terminal if it appears on the left hand side of a production.

Terminal symbols come in three varieties: reserved words, lexer tokens, and special string tokens. Reserved word terminal symbols will appear in a bold-faced font. The HLA lexical analyzer returns certain string classes (e.g., identifiers and constants) using a classification token (e.g., identifier or integer constant). We will treat these as terminal symbols for reasons of brevity (basically, we'll define terminal symbols in this grammar as the things that HLA's lexer returns to the parser). This grammar will use identifiers like intConst or identifier for these special terminal symbols. To differentiate these identifiers from non-terminals and reserved words, this document will underline such identifiers (e.g., identifier and intConst). Special string tokens (e.g., ":=") will be enclosed by apostrophies (e.g., ':='). This saves having to use less meaningful identifiers for these objects when their string form is much clearer. The special form ' (an underlined apostrophe) denotes an actual apostrophe token in the grammar.

A production in the grammar takes the following form:

leftHandSide → *right hand side*

The *leftHandSide* is always a single identifier (a non-terminal). On the right hand side is a string of one or more terminal, non-terminal, or meta-symbols. One example of a meta-symbol is 'ε' which represent the empty string; you'll see some additional meta-symbols in moment. Here is a typical production from the HLA grammar:

HLAPgm → **unit** identifier ';' unitDcls **end** identifier ';'

This says that an *HLAPgm* may consist of the reserved word UNIT followed by an identifier, followed by a semicolon, followed by some *unitDcls*, followed by an END, followed by an identifier, and ending with a semicolon.

To reduce the number of productions in the grammar, HLA v2.0's grammar uses some extensions. Like EBNF, HLA's grammar allows the use of the "|" meta-symbol to combine productions. Consider the following:

A → B | C

The production above is comletely equivalent to the following two productions:

A → B
A → C

HLA v2.0's grammar notation borrows some additional notation from regular languages. The parentheses, "*" and "+" symbols have their usual meaning from regular expressions (except, of course, they may control non-terminal symbols as well as terminal symbols. Consider the following production:

AddExpr → MulExpr ('+' MulExpr)*

This production says that an *AddExpr* consists of a *MulExpr* followed by zero or more occurrences of a '+' followed by a *MulExpr*. This is equivalent to the following set of productions:

AddExpr → MulExpr AddExpr2
AddExpr2 → ε
AddExpr2 → '+' AddExpr

Clearly, the former version is shorter, easier to read, and easier to understand. Hence the use of regular expression notation in HLA v2.0's grammar. Note that the following is also equivalent to the above:

AddExpr → MulExpr
AddExpr → AddExpr '+' MulExpr

Unfortunately, the second production above is left-recursive and must be converted to a right-recursive set of productions before you may directly implement the productions in a predictive recursive descent parser (like the one HLA v2.0 uses). Fortunately, the kleen star ("*") notation is readily convertable to code by simply using a WHILE loop. Hence, it is perfect for HLA's grammar.

HLA v2.0 Run-Time Language Grammar

Note: this grammar covers only the HLA run-time language (the assembly language). It does not deal with the compile-time language grammar (which appears elsewhere). Note that the HLA lexical analyzer automatically engages the compile-time language grammar as it encounters a lexeme associated with the compile-time language.

Another important thing to note is that this grammar was designed as an implementation tool for a recursive descent predictive parser. In some cases, readability was sacrificed in the productions in order to produce a grammar that could be directly implemented with code (in particular, left recursion has been removed, as necessary, and left factoring has been done; at least, as discovered in the grammar).

HLA v2.0 grammar start symbol: HLAPgm.

HLAPgm →
program identifier ';'
pgmDcls
begin identifier ';'
statements
end identifier ';'

Semantic issues: the three identifiers in this production must all be the same.

HLAPgm →
    unit identifier ';'
        unitDcls
    end identifier ';'

Semantic issues: the two identifiers in this production must both be the same.

pgmDcls → (pDcls)*
pDcls → **var** Variables
| **namespaceDcl** optionalSemicolon
| **const** Constants
| **val** Values
| **type** Types
| **static** StaticVars
| **storage** StorageVars
| **readonly** ReadonlyVars
| **segment** SegmentVars
| **procedure** procStuff
| **iterator** procStuff
| **method** procStuff
| **macro** macroStuff
| **template** templateStuff

unitDcls → (uDcls)*
uDcls → **namespaceDcl** optionalSemicolon
| **const** Constants
| **val** Values
| **type** Types
| **static** StaticVars
| **storage** StorageVars
| **readonly** ReadonlyVars
| **segment** SegmentVars
| **procedure** procStuff
| **iterator** procStuff
| **method** procStuff
| **macro** macroStuff
| **template** templateStuff

optionalSemicolon → ';' | ε
oneOrMoreSemicolons → (';')+

namespaceDcl →
    namespace nsID ';'
        NSdcls
    end nsID ';'

Semantic issue: the two identifiers above must be identical.

nsID → identifier

Semantic issue: If the identifier already exists, it must exist at lex level zero or one and it must be an existing namespace identifier..

NSdcls → (nDcls)*
nDcls → **const** Constants
| **val** Values
| **type** Types
| **static** StaticVars
| **storage** StorageVars
| **readonly** ReadonlyVars
| **segment** SegmentVars
| **procedure** procStuff
| **iterator** procStuff
| **method** procStuff
| **macro** macroStuff
| **template** templateStuff

Constants → (constDcls)*
constDcls → ';'
| ulID cDcls

cDcls → ':' ForC ';'
| ':=' constExpr ';'

ForC → typeID optionalbounds ':=' constExpr
| **enum** '{' idList '}' ':=' constExpr
| **pointer to** ptrTypeID ':=' constExpr
| **forward** '(' fID ')'

Semantic Issue: In the productions above, the type of *constExpr* must be compatible with the type. The fID identifier must be a legal VAL type identifier within this scope.

ulID → identifier

Semantic Issue: The identifier must be unique within the current (local) scope.

Values → (valDcls)*
valDcls → ';'
| vID vDcls

vDcls → ':' ForV ';'
| ':=' constExpr ';'

ForV → typeID optionalbounds optAssign
| **enum** '{' idList '}' optAssign
| **pointer to** ptrTypeID optAssign
| **forward** '(' fID ')'

Semantic Issue: In the productions of *ForV* above, the type of *constExpr* returned via *optAssign s constExpr* must be compatible with the type.

vID → identifier

Semantic issue: The identifier need not be unique within the current (local) scope. If it is unique at the current scope, then this declaration creates a new identifier at the current lex level; if the identifier is not unique at the local scope, then the *valDcls* will use the existing symbol. For forward declarations, *vID* must be unique.

dimList → constExpr (',' constExpr)*

Types → (typeDcls)*

typeDcls → ';'
| tID ':' tDcls

tDcls → typeID optionalbounds ';'
| **enum** '{' idList '} ';'
| **pointer to** ptrTypeID ';'
| **forward** '(' fID ')' ';'
| **record** recStuff ';'
| **union** recStuff ';'
| **class** classStuff ';'
| **procedure** optionalParms ';' protoOptions

optionalBounds → '[' dimList ']'
| ε

protoOptions → ε
| **returns** '(' constExpr ')' ';' protoOptions
| **pascal** ';' protoOptions
| **stdcall** ';' protoOptions
| **cdecl** ';' protoOptions

ptrTypeID → typeID

Semantic Issue: If *typeID* is undefined, the program must define it at some point within the current scope (i.e., at the current lex level). If *typeID* already exists at a global lex level, this declaration uses that definition, even if a local declaration appears later in the current scope. Note that HLA will enforce case neutrality, even if typeID exists only at a global scope.

typeID → identifier
→ builtInTypes

builtInTypes → **thunk**
| constBITypes

constTypes → **boolean**
| **uns8**
| **uns16**
| **uns32**

| **uns64**
| **uns128**
| **byte**
| **word**
| **dword**
| **qword**
| **tbyte**
| **lword**
| **int8**
| **int16**
| **int32**
| **int64**
| **int128**
| **char**
| **xchar**
| **unicode**
| **real32**
| **real64**
| **real80**
| **string**
| **ustring**
| **cset**
| **xcset**
| **text**

Semantic note: if the *typeID* is an identifier, the class of that identifier must be *Type_ct*. Note that TEXT objects are only legal in CONST and VAL declaration sections. THUNK objects are not legal in CONST and VAL declarations sections. If the *typeID* is an identifier, that identifier must be defined at the point HLA encounters *typeID* (unless otherwise specified, e.g., for **pointer** types). If *typeID* is a global symbol and the current scope later redefines *typeID*, then *typeID* uses the global definition in existence at the point *typeID* was encountered.

Variables → optionalVarAlign (vDcls)*

optionalVarAlign → ε
| '(' constExpr ')'
| ':=' constExpr ';'

vDcls → ';'
| vID ':' vTypes

vTypes → typeID optionalbounds ';'
| **align** '(' constExpr ')' ';'
| **enum** '{' idList '}' ';'
| **pointer to** ptrTypeID ';'
| **forward** '(' fID ')' ';'
| **record** recStuff ';'
| **union** recStuff ';'
| **class** classStuff ';'
| **procedure** optionalParms ';' protoOptions

StaticVars → optionalStaticAlign (stDcls)*

optionalStaticAlign → ε
| '(' constExpr ')'

stDcls → ';'
| **vmt** '(' typeID ')' ';'
| **align** '(' constExpr ')' ';'
| stID ':' stTypes

stTypes → **vmt** '(' typeID ')' ';'
| **align** '(' constExpr ')' ';'
| **forward** '(' fID ')' ';'
| **enum** '{' idList '} optAssign ';' varOptions
| typeID optionalbounds optInit
| **pointer to** ptrTypeID optInit
| **record** recStuff ';' varOptions
| **union** recStuff ';' varOptions
| **procedure** optionalParms procInit

optAssign → ':=' constExpr
| ε

optInit → ':=' constExpr ';'
| ';' varExtOptions

procInit → ':=' constExpr ';' protoOptions
| ';' protoOptions extOrNotsto

varOptions → ε
| **nostorage** ';' justVolatile
| **volatile** ';' varOptions
| **at** '(' constExpr ')' ';' justVolatile

justVolatile → **volatile** ';'
| ε

varExtOptions → **volatile** ';' extOrNosto
| **nostorage** ';' justVolatile
| **external** optExtStr ';' justVolatile
| **at** '(' constExpr ')' ';' justVolatile
| **forward** ';' justVolatile
| ε

extOrNosto → **nostorage** ';'
| **external** optExtStr ';' justVolatile
| **forward** ';' justVolatile
| **at** '(' constExpr ')' ';' justVolatile
| ε

optExtStr → '(' constExpr ')'
| ε

Syntax Issue: Note that the grammar allows both the NOSTORAGE and EXTERNAL options; the parser must handle this issue.

StorageVars → optionalStaticAlign (stoDcls)*

stoDcls → ';'
| **align** '(' constExpr ')' ';'
| stoID ':' stoTypes

stoTypes → **align** '(' constExpr ')' ';'
| **forward** '(' fID ')' ';'
| **enum** '{' idList '}' ';' varOptions
| typeID optionalbounds ';' varExtOptions
| **pointer to** ptrTypeID ';' varExtOptions
| **record** recStuff ';' varOptions
| **union** recStuff ';' varOptions
| **procedure** optionalParms ';' protoOptions extOrNotsto

ReadonlyVars → optionalStaticAlign (roDcls)*

roDcls → ';'
| **vmt** '(' typeID ')' ';'
| **align** '(' constExpr ')' ';'
| roID ':' roTypes

roTypes → **vmt** '(' typeID ')' ';'
| **forward** '(' fID ')' ';'
| **align** '(' constExpr ')' ';'
| **enum** '{' idList '} extNostoOrAssign
| typeID optionalbounds extNostoOrAssign
| **pointer to** ptrTypeID extNostoOrAssign
| **procedure** optionalParms ropOptions

extNostoOrAssign → ':=' constExpr ';'
| ';' extOrNotsto

ropOptions → ':=' constExpr ';' protoOptions
| ';' protoOptions extOrNotsto

SegmentVars → identifier '(' constExpr ',' constExpr ')' (stDcls)*

idList → identifier (',' identifier)*

optionalParms → ε
| '(' oneOrMoreParms ')'

oneOrMoreParms → passBy identifier ':' typeOrProc
| **var** identifier ':' varStuff

Semantic issues: if the VAR keyword appears in place of the type, then the object must be passed by reference. When passing a parameter in a register, the size of the register must match the size of the parameter (four bytes for reference, value/result, and result parameters; eight bytes for pass by name and pass by lazy evaluation parameters; object size for pass by value).

optionalIN → **in** register
| ε

passBy → **val**
| **valres**
| **name**
| **lazy**
| ε

typeOrProc → parmTypeID optionalIN (';' oneOrMoreParms)*
| **procedure** optionalParms (';' oneOrMoreParms)*

varStuff → **var** optionalIN (';' oneOrMoreParms)*
| parmTypeID optionalIN (';' oneOrMoreParms)*
| **procedure** optionalParms optionalIN (';' oneOrMoreParms)*

register → reg32 | reg16 | reg8

reg32 → **eax | ebx | ecx | edx | esi | edi | ebp | esp**

reg16 → **ax | bx | cx | dx | si | di | bp | sp**

reg8 → **al | ah | bl | bh | cl | ch | dl | dh**

recordDcl → optionalInherits recunVars privateRecVars **endrecord**

unionDcl → recunVars **endunion**

optionalInherits → ε
| inherits '(' identifier ')'

recunVars → ruVars (ruVars)*

privateRecVars → ε
| **private** ':' recunVars

ruVars → **align** '(' constExpr ')' ';'
| **record** recunVars **endrecord** ';'
| **union** recunVars **endunion** ';'
| orID ':' ruDcls

ruDcls → **forward** '(' constExpr ')'
| **record** recordDcl ';'
| **union** unionDcl ';'
| **procedure** optionalParms protoOptions ';'
| **enum** '{' idList '}' ';'
| **pointer to** typeID ';'
| typeID optionalBounds';'

orID → identifier
| **override** identifier

Semantic issue: OVERRIDE is only legal in records that have inherited fields from another record. The identifier following the OVERRIDE keyword must have been a field in the base record.

classDcl → **class** optionalInherit (clsDcls)$^{+}$ **endclass** ';'

clsDcls → **const** Constants
| **val** Values
| **type** Types
| **var** Variables
| **static** StaticVars
| **storage** StorageVars
| **readonly** ReadonlyVars
| **procedure** procProto
| **iterator** procProto
| **method** procProto
| **macro** macroStuff
| **template** templateStuff

procStuff → pID optionalParms ';' protoOptions extOrBody

extOrBody → **external** optExtStr ';'
| **forward** ';'
| procDcls
begin pID ';'
statements
end pID ';'

pID → identifier

Semantic Note: *pID* in the *extOrBody* production must match the *pID* in the *procStuff* production.

procDcls → (prDcls)*
prDcls → **var** Variables
| **const** Constants
| **val** Values
| **type** Types
| **static** StaticVars
| **storage** StorageVars
| **readonly** ReadonlyVars
| **procedure** procStuff
| **iterator** procStuff
| **method** procStuff
| **macro** macroStuff
| **template** templateStuff

Syntax Note: SEGMENTS and NAMESPACES are illegal in procedures, iterators, and methods.

procProto → prID optionalParms ';' protoOptions optionalExtFwd

prID → identifier

optionalExtFwd → ε
| **external** optExtStr ';'
| **forward** ';'

macroStuff → mID optionalMacroParms optionalMacroLocals ';' macroBody

mID → identifier

optionalMacroParms → '(' mpID (',' mpID)* ')'
| ε

mpID → identifier

optionalMacroLocals → ':' idList
| ε

macroBody → **keyword** keywdStuff
| **terminator** termStuff
| **endmacro** ';'
| << any textual character/token not matching one of the above>> macroBody

KeywdStuff → mID optionalMacroParms optionalMacroLocals ';' keywdBody

keywdBody → **keyword** keywdStuff

| | **terminator** termStuff
| | << any textual character/token not matching one of the above>> macroBody

termStuff → mID optionalMacroParms optionalMacroLocals ';' termBody

termBody → **endmacro** ';'
| | << any textual character/token not matching one of the above>> termBody

constExpr → expr

Semantic Issue: *expr* must return a "constant" classification in the production above.

memory → expr

Semantic Issue: expr must return a "memory" classification in the production above.

term → '(' constOrType ')'

Semantic Issue: The production above returns "constant" or "memory" as returned by *constOrType*.

term → litConstant

Semantic Issue: The expression above returns "constant" as its classification.

term → optSegReg xID optIndex

Semantic Issue: The production above returns "constant" or "memory" based on what *xID* returns. If *optIndex* returns memory, *xID* must also have the "memory" classification. If *optIndex* is "constant" then *xID* may be "constant" or "memory". If *xID* is not a static object, then *optIndex* may not return both a base and index register (since non-static objects already use a base pointer register).

terrm → optSegReg index

Semantic Issues: This production always returns the "memory" classification, even if index returns "constant" (this would correspond to an operand like "[$1234567]" that we want to treat as a memory operand).

term → '-' constExpr

Semantic Issue: The expression above returns "constant" as its classification. The *constExpr* type can be any numeric type or a character set type.

term → '!' constExpr

Semantic Issue: The expression above returns "constant" as its classification. The *constExpr* type can be any integer type or boolean. (This is the logical not operator, zero/not zero.)

term → '~' constExpr

Semantic Issue: The expression above returns "constant" as its classification. The *constExpr* type can be any integer type or boolean. (This is the bit inversion operator. For booleans, same as '!'.)

term → constFuncs

Semantic Issue: The expression above returns "constant" as its classification.

mulp → term (mulOp term)*

mulOp → '*' | '/' | **div** | **mod** | "<<" | ">>"

Semantic Issue: The *term* items must have a "constant" classification unless the *mulp* item consists of a single *term*. The *mulp* production returns the "constant" classification unless there is a single *term*, in which case *mulp* returns whatever classification *term* returns.

addp → mulp (addOp mulp)*

addOp → '+' | '-'

Semantic Issue: The *mulp* items must have a "constant" classification unless the *addp* item consists of a single *mulp* item. The *addp* production returns the "constant" classification unless there is a single *mulp*, in which case *addp* returns whatever classification *mulp* returns.

cmpp → addp (cmpOp addp)*

cmpOp → '<' | '<=' | '=' | '==' | '<>' | '!=' | '>=" | '>' | **in**

Semantic Issue: The *addp* items must have a "constant" classification unless the *cmpp* item consists of a single *addp* item. The *cmpp* production returns the "constant" classification unless there is a single *addp*, in which case *cmpp* returns whatever classification *addp* returns.

andp → cmpp ('&' cmpp)*

Semantic Issue: The *cmpp* items must have a "constant" classification unless the *andp* item consists of a single *cmpp* item. The *andp* production returns the "constant" classification unless there is a single *cmpp*, in which case *andp* returns whatever classification cmpp returns.

expr → andp (orOp andp)*

orOp → '|' | '^'

Semantic Issue: The *andp* items must have a "constant" classification unless the *expr* item consists of a single *andp* item. The *expr* production returns the "constant" classification unless there is a single *andp*, in which case *expr* returns whatever classification andp returns.

expr → '&' anyStaticID optStaticBounds
→ '&' anyStaticID optStaticBounds '-' '&' anyStaticID optStaticBounds

Semantic Issue: These productions always return the classification "constant".

anyStaticID → identifier

Semantic Issue: The identifier must be a STATIC, READONLY, STORAGE, SEGMENT VARIABLE, PROCEDURE, METHOD, or ITERATOR identifier.

optStaticBounds → ε
| '[' constExpr ']'

Semantic Issue: the *constExpr* must be a 32-bit integer constant value.

constOrType → **type** typeID expr
| constExpr

Semantic Issue: If the *constExpr* form is used, this production returns "constant" as its classification. If the "type typeID expr" form is used, then this production returns the classification that the *expr* returns.

litConstant → intConst
| fltConst
| charConst
| strConst
| arrayConst
| recordConst
| csetConst

Semantic Issue: The expressions above return "constant" as their classifications.

arrayConst → '[' constExpr (',' constExpr)* ']'

Semantic Issue: The expression above returns "constant" as its classification.

recordConst → typeID ':' '[' constExpr (',' constExpr)* ']'

Semantic Issues: The expression above returns "constant" as its classification. *typeID* must be defined and it's class must be *Type_ct* and it's *pType* must be *Record_pt*. The number of constant expressions must match the number of fields specified by *typeID*.

```
csetConst   →  '{' csetItem ( ',' csetItem )*  '}'

csetItem    →  charConst
            |  charConst '..' charConst
```

Semantic Issue: The expressions above return "constant" as their classifications.

```
xID         →  optBase identifier ( '.' identifier )*
```

Semantic Issue: If identifier is a CONSTANT or VAL object, then this production returns "constant" as it's classification. If identifier is a VAR, STATIC, READONLY, STORAGE, or SEGMENT object, then this production returns "memory" as it's classification. If identifier is none of these, then we have an error. If the "dot path" is present, the dotted names must select fields in a record, union, class, or namespace from the base identifier. If the OPTBASE is present, then the identifier must be a non-local, non-static, variable.

```
optBase     →  ε
            |  reg32 '::'

index       →  '[' baseOrConst
optIndex    →  index
            |  ε
```

Semantic Issues: The *index* production returns whatever classification ("constant" or "memory") that *baseOrConst* returns. The *optIndex* production returns whatever *index* returns or "constant" for the empty string (with a displacement of zero).

```
baseOrConst →  reg32  endScaleOrConst
            |  constExpr ']' optIndex
```

Semantic Issues: The production beginning with *reg32* always returns "memory" as its classification. The second production returns whatever classification that *optIndex* returns plus the displacement value specified by the *constExpr*.

```
endScaleOrConst   →  ']' optIndex
                  |  '+' indexOrConst
```

Semantic Issue: These two productions always return "memory" as their classification. If optIndex returns a base register, then the parser converts it to an index register with a scale value of "*1".

```
indexOrConst   →  reg32 Scale endOrConst
               |  constExpr ']' optIndex
```

Semantic Issues: If *endOrConst* returns a register (base or index) then this is an error. The first production above returns an index register with the scale factor specified by *Scale*. The second production returns the attributes of the *optIndex* item plus the displacement value specified by the *constExpr* item.

```
endOrConst  →  '+' constExpr ']' optIndex
            |  ']' optIndex
```

Semantic Issue: These two productions return whatever attributes *optIndex* returns (plus the displacement specified by the first production, if it gets used).

```
scaled      →  '*' constExpr
```

Semantic Issues: The *constExpr* above must be one of the following integer constants: 1, 2, 4, or 8.

```
constFuncs  →  constTypes '(' constExpr ')'
```

Semantic Issues: The type that *constExpr* returns must be convertable to the type specified by the *constTypes* item.

```
constFuncs      →  p1NumFuncs '(' constExpr ')'
p1NumFuncs      →  @abs | @ceil | @cos | @exp | @floor | @log | @log10 | @sin | @sqrt | @tan
```

Semantic Issues: *constExpr* must have a numeric attribute. Except for *@abs*, these functions all return a real constant. *@abs* returns a value whose type is the same as the parameter.

```
constFuncs      →  p1IntFuncs '(' constExpr ')'
p1NumFuncs      →  @odd | @random | @randomize
```

Semantic Issues: *constExpr* must have an integer attribute. @ODD returns a boolean value, the other two return integer values.

```
constFuncs      →  p1CharFuncs '(' constExpr ')'
p1NumFuncs      →  @isAlpha | @isAlphaNum | @isDigit | @isLower | @isSpace
                |  @isUpper | @isXdigit
```

Semantic Issues: *constExpr* must have a character or string attribute. These functions all return a boolean attribute.

```
constFuncs      →  @byte '(' constExpr  ',' constExpr')'
```

Semantic Issues: The value of the first expression must be a scalar data type, the second parameter must be a small integer value.

```
constFuncs      →  @date
                |  @time
```

Semantic Issues: These two functions always return a string constant.

constFuncs → **@extract** '(' constExpr optDestCS ')'

Semantic Issues: This function requires a character set parameter. It returns a character constant value. If the optional character set identifier is present, this function stores the resulting character set (without the extracted character) into the destination CSET variable.

optDestCS → ',' csID
| ε

csID → identifier

Semantic Issues: *csID* must be a const or val character set object.

constFuncs → minOrMax '(' constExpr (',' constExpr)* ')'
minOrMax → **@min | @max**

Semantic Issues: This functions require a list of constant expressions whose types are all the same or trivially convertable to the same type (e.g., integer to real). Allowable types: unsigned, integer, real, character, boolean, or enumerated types. These functions return a constant that is the same as the parameter type.

constFuncs → siiStrFuncs '(' $constExpr_1$ ',' $constExpr_2$ ',' $constExpr_2$ ')'
siiStrFuncs → **@delete | @substr**

Semantic Issues: $constExpr_1$ must be a string expression, $constExpr_2$ must be an integer expression. These functions return a string result.

constFuncs → indexFuncs '(' $constExpr_1$ ',' $constExpr_2$ ',' $constExpr_1$ ')'
indexFuncs → **@index | @rindex**

Semantic Issues: $constExpr_1$ must be a string expression, $constExpr_2$ must be an integer expression. These functions return a string result.

constFuncs → **@insert** '(' $constExpr_1$ ',' $constExpr_2$ ',' $constExpr_1$ ')'

Semantic Issues: $constExpr_1$ must be a string expression, $constExpr_2$ must be an integer expression. This function returns a string result.

constFuncs → **@length** '(' constExpr ')'

Semantic Issues: *constExpr* must be a string expression. This function returns an integer result.

constFuncs → caseFuncs '(' $constExpr_1$ ',' $constExpr_2$ ')'
caseFuncs **@lowercase | @uppercase**

Semantic Issues: $constExpr_1$ must be a string expression, $constExpr_2$ must be an integer expression. These functions return a string result.

constFuncs → csFuncs '(' constExpr$_1$ ',' constExpr$_2$ ',' constExpr$_3$ ')'
csFuncs **@strbrk | @strspan**

Semantic Issues: *constExpr$_1$* must be a string expression, *constExpr$_2$* must be an integer expression, *constExpr$_3$* must be a character set value. These functions return an integer result.

constFuncs → **@strset** '(' constExpr$_1$ ',' constExpr$_2$ ')'

Semantic Issues: *constExpr$_1$* must be a string expression, *constExpr$_2$* must be an integer expression. This function returns a string result.

constFuncs → **@tokenize** '(' constExpr$_1$ ',' constExpr$_2$ ',' constExpr$_3$ ',' constExpr$_3$ ')'

Semantic Issues: *constExpr$_1$* must be a string expression, *constExpr$_2$* must be an integer expression, and *constExpr$_3$* must be a character set expression. This function returns an array of strings result.

constFuncs → **@trim** '(' constExpr$_1$ ',' constExpr$_2$ ')'

Semantic Issues: *constExpr$_1$* must be a string expression, *constExpr$_2$* must be an integer expression. This function returns a string result.

constFuncs → **@linenumber | @curlex | @curoffset | @curdir | @add1stoffs | @section**
| **@parmoffset | @localoffset | @enumsize**

Semantic Issues: These functions return an integer result.

constFuncs → **@filename | @lastobject | @curobject**

Semantic Issues: These functions return a string result.

constFuncs → **@bound | @into | @trace | @exceptions | @optstrings**

Semantic Issues: These functions return a boolean result.

Pattern matching compile-time functions
---

constFuncs → cs2funcs '(' constExpr$_1$ ',' constExpr$_2$ optDests ')'
cs2funcs → **@peekcset | @onecset | @uptocset | @zeroormorecset | @oneormorecset**

Semantic Issues: *constExpr$_1$* must be a string expression, *constExpr$_2$* must be a character set expression. These functions all return true or false.

constFuncs → cs3funcs '(' constExpr$_1$ ',' constExpr$_2$ ',' constExpr$_3$ optDests ')'
cs3funcs → **@exactlyncset | @firstncset | @norlesscset | @normorecset**

Semantic Issues: *constExpr*$_1$ must be a string expression, *constExpr*$_2$ must be a character set expression, and *constExpr*$_3$ must be an integer expression. These functions all return true or false.

constFuncs → cs4funcs '(' constExpr$_1$ ',' constExpr$_2$ ',' constExpr$_3$ ',' constExpr$_3$ optDests ')'
cs4funcs → **@exactlyncset | @firstncset | @norlesscset | @normorecset**

Semantic Issues: *constExpr*$_1$ must be a string expression, *constExpr*$_2$ must be a character set expression, and *constExpr*$_3$ must be an integer expression. These functions all return true or false.

constFuncs → c2funcs '(' constExpr$_1$ ',' constExpr$_2$ optDests ')'
c2funcs → **@peekchar | @onechar | @uptochar | @zeroormorechar | @oneormorechar**
| **@peekichar | @oneichar | @uptoichar | @zeroormoreichar | @oneormoreichar**

Semantic Issues: *constExpr*$_1$ must be a string expression, *constExpr*$_2$ must be a character expression. These functions all return true or false.

constFuncs → c3funcs '(' constExpr$_1$ ',' constExpr$_2$ ',' constExpr$_3$ optDests ')'
c3uncs → **@exactlynchar | @firstnchar | @norlesschar | @normorechar**
| **@exactlynichar | @firstnichar | @norlessichar | @normoreichar**

Semantic Issues: *constExpr*$_1$ must be a string expression, *constExpr*$_2$ must be a character expression, and *constExpr*$_3$ must be an integer expression. These functions all return true or false.

constFuncs → c4funcs '(' constExpr$_1$ ',' constExpr$_2$ ',' constExpr$_3$ ',' constExpr$_3$ optDests ')'
c4funcs → **@exactlynchar | @firstnchar | @norlesschar | @normorechar**
| **@exactlynichar | @firstnichar | @norlessichar | @normoreichar**

Semantic Issues: *constExpr*$_1$ must be a string expression, *constExpr*$_2$ must be a character expression, and *constExpr*$_3$ must be an integer expression. These functions all return true or false.

constFuncs → s2funcs '(' constExpr$_1$ ',' constExpr$_1$ optDests ')'
s2funcs → **@matchstr | @uptostr | @matchtostr**
| **@matchistr | @uptoistr | @matchtoistr**

Semantic Issues: *constExpr*$_1$ must be a string expression. These functions all return true or false.

constFuncs → misc2PatFuncs '(' constExpr$_1$ optDests ')'
s2funcs → **@matchID | @matchIntConst | @matchRealConst**
| **@matchNumericConst | @matchStrConst | @matchReg**
| **@matchReg8 | @matchReg16 | @matchReg32**
| **@matchRegFPU | @matchRegMMX | @matchRegSSE**

Semantic Issues: *constExpr*$_1$ must be a string expression. These functions all return true or false.

constFuncs → misc1PatFuncs '(' constExpr optDest ')'
misc1PatFuncs → **@zeroormorews | @oneormorews | @wsoreos | @wstheneos**
| **@peekws | @eos**

Semantic Issues: *constExpr* must be a string expression. These functions all return true or false.

optDest → ε
| ',' identifier

Semantic Issues: The *identifier* operand must be a CONST or VAL string identifier

optDests → ε
| ',' identifier optDest

Semantic Issues: The *identifier* operand must be a CONST or VAL string identifier

optSegReg → **cs** ':'
| **ds** ':'
| **es** ':'
| **fs** ':'
| **gs** ':'
| **ss** ':'

todo:

templateStuff
statements