Introduction

Overview

Several independent tasks have to be performed during semantic analysis:

- Declaration Analysis
  - Go through and check the legality of the declarations (types, variables, procedures, etc) in the program. Check for:
    - multiple declarations: VAR x,y,z : int.
    - undeclared types: VAR x : integer.
    - illegal symbol kinds: VAR X : integer; VAR Y : X.
  - Construct symbol tables and environments to be used during name and expression analysis.

- Name Analysis
  - Match up each use of an identifier with the correct declaration. Report any undeclared identifiers.

- Expression Analysis
  - Assign types to every sub-expression.
  - Report type mismatches: X:="Hi" * "there".

- Prepare for Code Generation
  - Insert explicit type conversions: X:=5+6.7 ⇒ X:=float(5)+6.7.
  - Compute labels for conditional- and loop-statements.
Name Analysis

Algol-like languages allow the same name to be declared in different scopes. During name analysis the use of a name is matched up with the corresponding declaration.

```plaintext
VAR I, K : INTEGER;
PROCEDURE P (K : INTEGER);
    PROCEDURE Q (L : INTEGER);
        BEGIN I := L + K; (* I_{global} := L_Q + K_P *) END Q;
    VAR J : INTEGER;
    BEGIN I := J + K; (* I_{global} := J_P + K_P *) END P;
```

Symbol Tables

All compilers use a symbol table. In it we record all information regarding every declared item (variable, procedure, constant, etc).

The symbol table is built during declaration analysis.

The symbol table is used during name analysis and type checking to look up identifiers.

Some compilers build one (huge) symbol table for the entire input program. Others build one separate symbol table for each scope.

The type of information which is stored in a symbol table node depends on the declaration:
Symbol Tables...

All Symbols Name, Position, Level, Enclosing Block, ...

Variables Type, Size, ...

Constants Type, Size, Value, ...

Types Size, ...

Records Fields

Arrays Index-Type, Index-Range,
  Element-Type

Procedures Formal Parameters, Size of Local Data, ...

TYPE T = RECORD a, b : CHAR END;
VAR X : T;
PROCEDURE Q (a: T); BEGIN ... END Q;

Example — Building the Symbol Table

This time we want to build a symbol table from a sequence of declarations. At the same time we want to check for multiply declared identifiers.

In this example, the symbol table is simply a set of identifiers. Normally, we’d also want the symbol table to include other kinds of information: type, size, and offset for variables, formal parameters for procedures, etc.

Operations on SyTabs: Union (∪) and member (∈).

We use a threaded attribute Ids of type SyTab={String} (set of strings).

A threaded attribute Attr is really a combination of two attributes, a synthesized attribute AttrOut and an inherited attribute AttrIn.

Threaded attributes are used to collect information from a subtree. The evaluator uses the inherited attribute to collect information on the way down the tree, and the synthesized attribute to move that information back up the tree.

In our example, IdsIn is inherited and holds the current set of identifiers. IdsOut is synthesized and returns the complete symbol table.

Threaded Attributes

We use a threaded attribute Ids of type SyTab={String} (set of strings).

A threaded attribute Attr is really a combination of two attributes, a synthesized attribute AttrOut and an inherited attribute AttrIn.

Threaded attributes are used to collect information from a subtree. The evaluator uses the inherited attribute to collect information on the way down the tree, and the synthesized attribute to move that information back up the tree.

In our example, IdsIn is inherited and holds the current set of identifiers. IdsOut is synthesized and returns the complete symbol table.
Building the Symbol Table...

- At any node \( n \), \( n.\text{IdsOut} - n.\text{IdsIn} \) is the set of variables declared in the subtree rooted at \( n \).

--- Abstract Syntax: ---

\[
\text{Program} ::= \leftarrow \text{Id: String} \ \text{DeclSeq: Decl} \ \text{StatSeq: Stat} \\
\text{Decl} ::= \text{VarDecl} \mid \text{ProcDecl} \mid \text{NoDecl} \\
\text{VarDecl} ::= \leftarrow \text{Id: String} \ \leftarrow \text{TypeName: String} \ \text{Next: Decl} \\
\text{ProcDecl} ::= \leftarrow \text{Id: String} \ \text{Locals: Decl} \ \text{StatSeq: Stat} \ \text{Next: Decl} \\
\text{NoDecl} ::= \downarrow \text{Ids: SyTab}
\]

\[
\text{VarDecl} ::= \leftarrow \text{Id: String} \ \leftarrow \text{TypeName: String} \ \text{Next: Decl} \ \uparrow \text{Ids: SyTab} \\
\text{ProcDecl} ::= \leftarrow \text{Id: String} \ \text{Locals: Decl} \ \text{StatSeq: Stat} \ \text{Next: Decl} \ \uparrow \text{Ids: SyTab} \\
\text{NoDecl} ::= \downarrow \text{Ids: SyTab} \\
\]

\[
\text{Decl} ::= \uparrow \text{Ids: SyTab} \\
\]

\[
\text{PROCEDURE Program (n: Node);} \\
\quad \text{n.DeclSeq.IdsIn:={} }; \text{Decl(n.DeclSeq);} \\
\]

\[
\text{PROCEDURE Decl (n: Node);} \\
\quad \text{IF n.Kind } \in \{ \text{VarDecl, ProcDecl} \} \ \text{THEN} \\
\quad \quad \text{IF n.Id } \in \text{n.IdsIn} \ \text{THEN} \\
\quad \quad \quad \text{PRINT n.Pos "\\n:Multiple "declaration: " n.Id;} \\
\quad \quad \text{ENDIF} \\
\quad \quad \text{n.Next.IdsIn := n.IdsIn } \cup \{ \text{n.Id}; \\
\quad \quad \text{Decl(n.Next);} \\
\quad \quad \text{n.IdsOut := n.Next.IdsOut} \\
\quad \text{ELSIF n.Kind = NoDecl THEN} \\
\quad \quad \text{n.IdsOut := n.IdsIn} \\
\quad \text{ENDIF}
\]
Obviously, we don’t write one tree-walk evaluator for each attribute. Rather, we walk over the tree once (or maybe twice or three times, depending on the language) and evaluate as many attributes as possible.

In this example, we’ll just use several attribute evaluation rules in order to:

1. Check for multiple declarations and build a symbol table containing all the identifiers.
2. Assign an offset to each variable and compute the total size of the variables declared.
3. Assign a unique number to each declared identifier and count the number of identifiers.

---

**Abstract Syntax:**

Program ::= \( \leftarrow \text{Id}: \text{String} \\mid \text{Decl} \\mid \text{Stat} \)

Decl ::= \( \text{VarDecl} \mid \text{ProcDecl} \mid \text{NoDecl} \)

VarDecl ::= \( \leftarrow \text{Id}: \text{String} \\leftarrow \text{TypeName}: \text{String} \\mid \text{Next}: \text{Decl} \)

ProcDecl ::= \( \leftarrow \text{Id}: \text{String} \\mid \text{Args}: \text{Decl} \\mid \text{Locals}: \text{Decl} \\mid \text{Stat}: \text{Stat} \\mid \text{Next}: \text{Decl} \)

NoDecl ::= \( \text{Ids}: \text{SyTab} \mid \text{Count}: \text{INTEGER} \mid \text{Size}: \text{INTEGER} \)

---

**PROCEDURE Program (n: Node):**

\( \text{n.DeclSeq.IdsIn} := \{\}; \)
\( \text{n.DeclSeq.CountIn} := 0; \)
\( \text{n.DeclSeq.SizeIn} := 0; \)
\( \text{Decl(n.DeclSeq)}; \)

**PROCEDURE Decl (n: Node):**

IF \( \text{n.Kind} = \text{VarDecl} \) THEN \text{VarDecl(n)};
ELSIF \( \text{n.Kind} = \text{ProcDecl} \) THEN \text{ProcDecl(n)};
ELSIF \( \text{n.Kind} = \text{NoDecl} \) THEN

\( \text{n.CountOut} := \text{n.CountIn}; \)
\( \text{n.SizeOut} := \text{n.SizeIn}; \)
\( \text{n.IdsOut} := \text{n.IdsIn} \)
ENDIF
PROCEDURE VarDecl (n: Node);
    n.Next.SizeIn := n.SizeIn + size(n.TypeName);
    n.Next.CountIn := n.CountIn + 1;
    IF n.Id \in n.IdsIn THEN
        PRINT n.Pos ":Multiple declaration: " n.Id;
    ENDIF;
    n.Next.IdsIn := n.IdsIn \cup \{n.Id\};
    Decl(n.Next);
    n.SizeOut := n.Next.SizeOut;
PROCEDURE ProcDecl (n: Node);
    n.Next.SizeIn := n.SizeIn;
    (* Rest is same as for VarDecl *)

Putting it all together...
PROCEDURE Decl (n: Node);
IF n.Kind = VarDecl THEN
    Sy := (Name=n.Id, No=n.CountIn, Offset=n.SizeIn,
            Kind=VAR, Size=size(n.TypeName), Type=n.TypeName);
    n.Next.IdsIn := n.IdsIn ∪ {Sy};
    Decl(n.Next);
ELSIF n.Kind = ProcDecl THEN
    Sy := (Name=n.Id, Kind=PROC, No=n.CountIn);
    n.Next.IdsIn := n.IdsIn ∪ {Sy};
    Decl(n.Next);
ELSIF n.Kind = NoDecl THEN
    n.IdsOut := n.IdsIn
ENDIF

Implementing Symbol Tables

Symbol tables are sets of tuples. Any set data structure will do fine. Hash tables, binary search trees, or linked lists will be OK, depending on the size of the table.

TYPE KindT = (Var,Proc,Type,Const);
DataT = RECORD
    Name: String; Number: INTEGER; Pos: Position;
    CASE Kind : KindT OF
        Var : Type:String; ⋯ |
        Const : Value:INTEGER; ⋯ |
    END;
    Next : SyTab;
END;
SyTab = POINTER TO DataT;
This time we are going to build one symbol table for each nested scope.

Note that the formal parameters and local variables of a procedure belong to the same scope.

The next slide shows the abstract syntax for a language with variable and procedure declarations. The following slides show an example program, the attribute grammar, and the tree-walker.

Program ::= "Id":String DeclSeq:Decl StatSeq:Stat
Decl ::= VarDecl | ProcDecl | NoDecl
VarDecl ::= "Id":String "\leftarrow" TypeName:String Next:Decl
\uparrow Ids:SyTab
Formal ::= "Id":String "\leftarrow" TypeName:String Next:Decl
\uparrow Ids:SyTab
ProcDecl ::= "Id":String Args:Decl Locals:Decl Formals:Decl
StatSeq:Stat Next:Decl \uparrow Ids:SyTab
NoDecl ::= \uparrow Ids:SyTab

PROGRAM M;
PROCEDURE P ();
VAR X : INTEGER;
PROCEDURE Q (X : CHAR;
Z : INTEGER);
VAR Y : INTEGER;
VAR Z : CHAR;
BEGIN END Q;
BEGIN END P;
BEGIN END M;
Formal, VarDecl ::= \texttt{Id:String} \texttt{\Rightarrow TypeName:}String
Next:Decl \uparrow \texttt{Ids:SyTab}
{ Next.IdsIn := IdsIn \cup \{(Name=Id,Kind=VAR,\ldots)\};
IdsOut:=Next.IdsOut }

ProcDecl ::= \texttt{Id:}String Locals:Decl
Formals:Decl StatSeq:Stat Next:Decl \uparrow \texttt{Ids:SyTab}
{ Formals.IdsIn := \{\};
Locals.IdsIn := Formals.IdsOut;
Next.IdsIn := IdsIn \cup
\{(Name=Id,Kind=PROC,Vars=Locals.IdsOut,\ldots)\};
IdsOut:=Next.IdsOut }

NoDecl ::= \uparrow \texttt{Ids:SyTab} \{ IdsOut := IdsIn \}

PROCEDURE Decl (n: Node);
IF n.Kind = VarDecl THEN
Sy := (Name=n.Id,Kind=VAR,\ldots);
n.Next.IdsIn := n.IdsIn \cup \{Sy\};
Decl(n.Next); n.IdsOut:=n.Next.IdsOut
ELSIF n.Kind = ProcDecl THEN
n.Formals.IdsIn := \{\}; \texttt{NEW!}
Decl(n.Formals); \texttt{NEW!}
n.Locals.IdsIn := n.Formals.IdsOut;
Decl(n.Locals); \texttt{NEW!}
Sy := (Name=n.Id,Kind=PROC,\ldots);
n.Next.IdsIn := n.IdsIn \cup \{Sy\};
Decl(n.Next); n.IdsOut:=n.Next.IdsOut
ELSIF n.Kind = NoDecl THEN n.IdsOut := n.IdsIn ENDIF

Environments
Building Environments

Environments are used to represent scope information. They are linked lists of symbol tables where each symbol table represents the identifiers declared in a given scope.

First a symbol table is built for each scope. Then environments are constructed and sent down to statements to be used during name analysis.

\[ \text{EnvT} = \text{LIST OF SyTab} = \text{LIST OF SET OF Symbol}. \]

Abstract Syntax:

**VarDecl** ::= \( \leftarrow \text{Id: String} \leftarrow \text{TypeName: String} \) Next:Decl
\( \uparrow \text{Ids: SyTab} \downarrow \text{Env: EnvT} \)

**Formal** ::= \( \leftarrow \text{Id: String} \leftarrow \text{TypeName: String} \) Next:Decl
\( \uparrow \text{Ids: SyTab} \downarrow \text{Env: EnvT} \)

**ProcDecl** ::= \( \leftarrow \text{Id: String} \) Locals:Decl Formals:Decl StatSeq:Stat
Next:Decl \( \uparrow \text{Env: EnvT} \uparrow \text{Ids: SyTab} \)

**Assign** ::= Des:Des Expr:Expr \( \downarrow \text{Env: EnvT} \)

**Program** ::= \( \leftarrow \text{Id: String} \) DeclSeq:Decl StatSeq:Stat
\{ 
DeclSeq.Env := \{INT, REAL, CHAR, TRUNC, FLOAT\};
DeclSeq.IdsIn := \{};
StatSeq.Env := cons(DeclSeq.IdsOut, DeclSeq.Env);
\}

**Formal, VarDecl** ::= \( \leftarrow \text{Id: String} \leftarrow \text{TypeName: String} \)
\( \rightarrow \text{Decl} \uparrow \text{Ids: SyTab} \downarrow \text{Env: EnvT} \)
\{ 
CHECK NOT member(Env, TypeName) 
⇒ ERROR("Ident not declared")
Next.Env := Env;
\}

**ProcDecl** ::= \( \leftarrow \text{Id: String} \) Locals:Decl Formals:Decl
StatSeq:Stat \( \rightarrow \text{Decl} \downarrow \text{Env: EnvT} \)
\{ 
Formals.IdsIn := \{};
Formals.Env := Locals.Env := Env;
Locals.IdsIn := Formals.IdsOut;
StatSeq.Env := cons(Locals.IdsOut, Env);
Next.Env := Env;
\}

**Assign** ::= \( \rightarrow \text{Env: EnvT} \)
\{ Des.Env := Expr.Env := Env; \}
PROCEDURE Program (n: Node);
    StdEnv := {INT, REAL, CHAR, TRUNC, FLOAT};
    n.DeclSeq.Env := StdEnv;
    n.DeclSeq.IdsIn := {};
    Decl(n.DeclSeq);
    n.StatSeq.Env := cons(n.DeclSeq.IdsOut, StdEnv);
    Stat(n.StatSeq);

PROCEDURE Decl (n: Node);
    IF n.Kind=VarDecl THEN VarDecl(n);
    ELSIF n.Kind=ProcDecl THEN ProcDecl(n);
    ELSIF n.Kind=NoDecl THEN (* Same *)
    ENDIF

PROCEDURE ProcDecl (n: Node);
    n.Formals.IdsIn := {};
    n.Formals.Env := n.Locals.Env := n.Env;
    Decl(n.Formals);
    n.Locals.IdsIn := n.Formals.IdsOut;
    Decl(n.Locals);
    n.StatSeq.Env := cons(n.Locals.IdsOut, n.Env);
    Stat(n.StatSeq);
    n.Next.Env := Env;
    (* More here...*)

PROCEDURE Assign (n: Node);
    Des.Env := Expr.Env := n.Env; ...

PROCEDURE VarDecl (n: Node);
    IF NOT member(n.Env, n.TypeName) THEN
        PRINT n.Pos " Identifier not declared " n.TypeName;
    ENDIF;
    n.Next.Env := Env;
    (* More here...*)
Implementing Environments

__________ Symbol Table ADT ___________

Create() : SyTabT;
Insert(S:SyTabT; N:Name) : SymbolRef;
Lookup(S:SyTabT; N:Name) : SymbolRef;

- Each symbol R has a set of attributes (Type, Size,...) that can be set/retrieved using operations Set<Attr>/Get<Attr>(S,R,A).

__________ Environment ADT ___________

Create() : EnvT;
Cons(S:SyTabT; E:EnvT) : Env;
Identify(E:EnvT; N:Name) : SymbolRef;
Member(E:EnvT; N:Name) : BOOLEAN;

- Cons(S,E) creates a new environment consisting of the symbol table S followed by the symbol tables of E.
- Identify(E, N) searches the symbol tables of E sequentially until a definition of the name N is found.

Symbol tables are sets of tuples (collections of data), environments are lists of symbol tables.

Implementing Environments...
Implementing Environments...

TYPE KindT = (Var, Proc, Type, Const);

SyTabT = POINTER TO RECORD
  CASE Kind : KindT OF · · · END;
  Name : String; Next : SyTabT;
END;

EnvT = POINTER TO RECORD
  SyTab : SyTabT; Next : EnvT;
END;

Summary

During declaration analysis we build symbol tables that will be used during name analysis.

A symbol table is a collection of information about the identifiers declared in a program. The kind of information that is stored for a particular identifier depends on its kind (variable, procedure, etc).

For every identifier we store its name, kind, and position (line and column number in the source code where the identifier is declared).

Readings and References

- Read Louden:
  Symbol Tables 295–313
  Note that Louden uses different algorithms for symbol-tables and environments than I do in this lecture.
- or read the Dragon book:
  Symbol Tables 429–438
  Nested Procedures 415–416
  Environments 438–440
A threaded attribute $\uparrow A:T$ actually consists of two attributes: an inherited attribute $\downarrow A_{\text{In}}:T$ and a synthesized attribute $\uparrow A_{\text{Out}}:T$. As we perform an inorder traversal of a subtree, $A_{\text{In}}$ collects information from the tree, and $A_{\text{Out}}$ brings it back up the tree.

Threaded attributes are used to gather information from a subtree. Since gathering information is exactly what we do when we build a symbol table, we use a threaded attribute $\uparrow \text{Ids}:\text{SyTabT}$ to construct the symbol table.

This symbol table can then be passed down the tree (using an inherited attribute $\downarrow \text{Env}:\text{SyTabT}$) during name analysis.

An inherited attribute is given a value before a recursive call is made:
\[
n.LOP.\text{Env} := n.\text{Env}; \text{Expr}(n.LOP);
\]

An synthesized attribute is given a value before a recursive call returns:
\[
\text{PROCEDURE Expr (n: Node);}
\]
\[
\quad \text{IF n.Kind = IntConst THEN n.Type := "INT";}
\]

For a threaded attribute pair, the inherited part is given a value before the recursive call is made and the synthesized parts is given a value after the call returns:
\[
n.\text{Next.IdsIn} := n.\text{IdsIn} \cup \{\text{Sy}\};
n.\text{Decl}(n.\text{Next});
n.\text{IdsOut} := n.\text{Next.IdsOut}
\]

Tree-walkers use environments to perform name analysis. An environment is a list of symbol tables, where each table consists of the symbols collected in a particular scope.

Environments are organized so that if they are searched sequentially from the start, we’ll always find the correct (most closely nested) identifier first.

Environments are passed down the tree (using inherited attributes) in order to inform lower level nodes about the context in which they occur.

Why do we have to store all information in the AST? Why can’t we just use one global symbol-table to keep all data about all symbols? Passing these environments around seems really inefficient and confusing. It is true that some compilers build one huge symbol table for the entire program and keep that outside the tree. This method works well for simple languages like C, which does not support nested procedures, classes, etc. For other languages, it’s better to build one symbol table for each scope, and pass them around the tree using attributes. Then we’ll have complete control of the information that is available at each point in the program; we’ll know exactly what information is passed into each node, and what attributes are computed at each node.
Show the environment in effect at each point \( i \) in this program. Identifiers must be declared before use. Recursion is allowed.

Program M:

```
TYPE T = ARRAY 5 OF CHAR; VAR X : INTEGER;
PROCEDURE P ();
    VAR Z : T;
    PROCEDURE Q ();
        VAR R : CHAR; PROCEDURE Z (X:CHAR); BEGIN 3 END Z
        VAR Y : CHAR; PROCEDURE V (); BEGIN 4 END V;
    BEGIN 5 END Q;
    VAR Y : INTEGER;
    BEGIN 6 END P;
    VAR Y : INTEGER;
    BEGIN 1 END M.
```

---

Homework I

Show the environment in effect at each point \( i \) in the program below.

Program M:

```
VAR X : INTEGER;
PROCEDURE P (X : CHAR);
    VAR Z : INTEGER;
    PROCEDURE Q (X : INTEGER);
        VAR R : CHAR;
        PROCEDURE Z (); BEGIN 1 END Z;
        VAR Y : CHAR;
        BEGIN 2 END Q;
        VAR Y : INTEGER;
        BEGIN 3 END P;
        VAR Y : INTEGER;
        BEGIN 4 END M.
```

Homework II

Show the symbol tables resulting from the declarations below. Include as much information about each symbol as possible. Give each identifier a unique number (set INTEGER=1 and CHAR=2), and use these numbers to represent types.

Problem (A): 

```
PROCEDURE P (X:INTEGER; Y:CHAR);
VAR Z:INTEGER;
VAR Y : INTEGER;
BEGIN END P;
```
Homework II

Problem (B):

TYPE T = RECORD A, B : CHAR END;
VAR X : T;

Problem (C):

TYPE T2 = POINTER TO CHAR;
TYPE T2 = ARRAY 100 OF T1;
TYPE T3 = ARRAY 20 OF T2;

Homework III

Build an abstract syntax tree for the program below. Show — in detail — how the symbol tables and environments are built.

PROGRAM M;
VAR X : INTEGER;
VAR Y : INTEGER;
PROCEDURE P (X : CHAR);
VAR Z : INTEGER;
PROCEDURE Q (X : INTEGER);
VAR R : CHAR;
VAR V : CHAR;
BEGIN END Q;
VAR Y : INTEGER;
BEGIN END P;
BEGIN END M.

Homework IV

Build an abstract syntax tree for the program below. Show — in detail — how the statements are type checked. Which error messages should be generated?

PROGRAM M;
VAR X : INTEGER;
VAR Y : INTEGER;
PROCEDURE P (Z : INTEGER; VAR X : CHAR);
VAR Z : INTEGER;
BEGIN
  X := "D";
  Y := Z + X;
END P;
BEGIN
  P(X, "C");
END M.

Homework V

Assume a small Modula-2 like language:

Concrete Syntax:

Block ::= BEGIN StatSeq END
AssignStat ::= ident ':=' Expr
ForStat ::= FOR ident ':=' Expr TO Expr [ByPart] DO StatSeq END
ByPart ::= BY ConstExpr
Stat ::= AssignStat | IfStat | ForStat
StatSeq ::= Stat ';' StatSeq | ε
Expr ::= Expr + Expr | ident | IntConst
Homework V.

1. Give an abstract syntax corresponding to the concrete syntax above.
2. Write a attribute grammar/tree-walk evaluator which checks that the `ByPart`, if present, is a constant expression.

Homework VI.

Assume a small Modula-2 like language like in the previous exercise, but with IF-statements:

Concrete Syntax Extension:

- `IfStat ::= IF Expr THEN StatSeq ELSE StatSeq END`  
- `IfStat ::= IF Expr THEN StatSeq END`  
- `ForStat ::= FOR ident ':=' expr TO expr [ByPart] DO StatSeq END`  
- `ByPart ::= BY ConstExpr`

Homework VII.

Assume that enumerated types are declared in this fashion:

```
TYPE T = ENUM[Marge=1, Bart=2,  
Maggie=5, Lisa=10];
```

I.e., unlike Pascal, we’re allowed to number the identifiers however we like.

1. Give an abstract syntax corresponding to the concrete syntax.
2. Write a attribute-grammar/tree-walk evaluator which checks that the iteration variable of a FOR-loop is not changed within the body of the loop. Remember that loops can be nested!

Assume that enumerated types are declared in this fashion:

```
TYPE T = ENUM[Marge=1, Bart=2,  
Maggie=5, Lisa=10];
```

I.e., unlike Pascal, we’re allowed to number the identifiers however we like.

1. Give a suitable abstract syntax and a tree-walk evaluator that checks that all identifiers and values are unique (within the declaration).
Homework VII

In other words, the static semantics should flag these declarations as erroneous:

TYPE T1 = ENUM[Ren=3, Stimpy=4, Ren=2];
“ERROR: Multiple enumeration id: Ren”

TYPE T2 = ENUM[CB=10, Linus=4, Lucy=10];
“ERROR: Repeated enumeration value: 10”

Homework VIII

Assume that enumerated types are declared in the “normal” Pascal fashion:

TYPE T = ENUM[Marge, Homer, Bart, Maggie, Lisa];

Assume furthermore that the individual identifiers are given numbers $0, 1, 2, \ldots$

Give a suitable abstract syntax and a tree-walk evaluator that computes the minimum number of bits required to store variables of the type.

Homework VIII

Examples:

TYPE T = ENUM[a] ⇒ 1 bit
TYPE T = ENUM[a,b] ⇒ 1 bit
TYPE T = ENUM[a,b,c] ⇒ 2 bits
TYPE T = ENUM[a,b,c,d] ⇒ 2 bits
TYPE T = ENUM[a,b,c,d,e] ⇒ 3 bits
TYPE T = ENUM[a,b,c,d,e,f] ⇒ 3 bits

Homework IX

1. Write a concrete and an abstract grammar for Pascal-like variable declarations.
2. Write a tree-walk evaluator that checks for multiple declarations of the same identifier.

Example 1 (Correct):

VAR x : CHAR; y,z,a,b : INTEGER; n,s : BOOLEAN;

Example 2 (Wrong):

VAR x, y, z, x, a : CHAR;

Example 3 (Wrong):

VAR x : CHAR; y,z,a,x : INTEGER; n,x : BOOLEAN;