

# User Defined Types

## Standard Functions and Procedures

```
Expression : ...  
           | IDENTSY LEFTPARENSY Expression  
           | RIGHTPARENSY  
           | ...  
           ;
```

- The standard functions are:
  - chr
  - ord
  - inc
  - dec

## Predefined Procedures and Functions and the Symbol Table

```
entertype(entertype(newid(strdup("CHR"))), char_type, Function);
entertype(entertype(newid(strdup("ORD"))), int_type, Function);
entertype(entertype(newid(strdup("INC"))), NULL, Procedure);
entertype(entertype(newid(strdup("DEC"))), NULL, Procedure);
```

## Record Reference

```
Designator: . . .
    | Designator DOTSY IDENTSY
    | {$$ = recordref($1, $3);}
    | . . .
    ;
```

## Code for Record Reference

```
EXPR *recordref(EXPR *e, char *s)
{ TYPE *typer;
  ID *f; EX_KIND kindr;
  int invr, offset, reg;

  ... /* Check that e represents a record. If not, issue a message and quit.*/
  ... /* Traverse the list of fields pointed to be e->ex_type->ty_form.ty_record */
  ... /* looking for one that has name s. If such a field is found, set */
  ... /* f to point to it, and set typer to f->id_type. */
  ... /* Otherwise, issue an appropriate message and quit. */

  /* Calculate the address of the field from the address of the record */
  kindr = e->ex_kind;
  invr = e->ex_invr;
  offset = f->id_addr;
```

## Code for Record Reference

```
if (offset != 0) /* see if you can figure out why */
{ switch (kindr)
  { case LocalV:
    case GlobalV:
      invr += offset;
      break;
    case LocalA:
      ... /* Generate code to load address into a register (reg). */
      ... /* Generate code to add the offset to the register. */
      kindr = RegisterA;
      invr = reg;
      break;
    case RegisterA:
      ... /* Generate code to add the offset to the register */
      break;
    default: /* should never arise */
      break;
  } /* switch */
} /* if */
free(s);
free(e);
return(newexpr(typer, kindr, invr));
} /* recordref */
```

# Array Reference

Designator : Designator LBRACKETSY ExpressionList  
RBRACKETSY

```
{ $$ = arrayref($1, $3); }
```

```
...
```

```
;
```

# Code for Array Reference

```
EXPR *arrayref (EXPR *base, EXPR *index)
{ TYPE *typer; EX_KIND kindr;
  int invr, reg, size, low;

  ... /* Check that base represents an array. If not issue a message and quit. */
  ... /* Check if the type of the array index of base is the same type as index. */
  ... /* If not issue an appropriate message and quit. */
  while ( more elements in index )
  {
    typer = base->ex_type->ty_form.ty_array.ElementType;
    size = typer->ty_size;
    low = base->ex_type->ty_form.ty_array.IndexType->ty_form.ty_subrange.min;
```

## Code for Array Reference

```
/* Multiply index and low bound by size */
if (size != 1) /* figure out why */
{
  low *= size;
  switch (index->ex_kind)
  {
    case Cons:
      index->ex_inv = index->ex_inv * size;
      break;
    default:
      ... /* Generate code to multiply index by size leaving result */
      ... /* in register reg. */
      index->ex_kind = RegisterV;
      index->ex_inv = reg;
      break;
  } /* switch */
} /* if */

if (base->ex_kind == LocalV || base->ex_kind == GlobalV) /* figure this out */
{
  base->ex_inv = base->ex_inv - low;
  low = 0;
} /* if */
```

## Code for Array Reference

```
/* Shift index by the value of low bound */
if (low != 0)
{
  switch (index->ex_kind)
  {
    case Cons:
      index->ex_inv = index->ex_inv - low;
      break;
    default:
      ... /* Generate code to subtract low from index leaving the */
      ... /* result in register reg. */
      index->ex_kind = RegisterV;
      index->ex_inv = reg;
      break;
  } /* switch */
} /* if */
```

## Code for Array Reference

```
/* Add the value of the index to the address of the array's first */
/* element. Special cases are considered to improve the code quality */
switch (base->ex_kind)
{ case GlobalV:
  switch (index->ex_kind)
  { case Cons:
    base->ex_inv = base->ex_inv + index->ex_inv;
    break;
  default:
    ... /* Generate code to add the address of base with the */
    ... /* value of index leaving the result in register reg. */
    base->ex_kind = RegisterA;
    base->ex_inv = reg;
    break;
  } /* switch */
break;
```

## Code for Array Reference

```
case LocalV:
switch (index->ex_kind)
{ case Cons:
  base->ex_inv = base->ex_inv + index->ex_inv;
  break;
default:
  ... /* Generate code to add the address of base with the */
  ... /* value of index leaving the result in register reg. */
  base->ex_kind = RegisterA;
  base->ex_inv = reg;
  break;
} /* switch */
break;
```

# Code for Array Reference

```
case RegisterA:
case LocalA:
  switch (index->ex_kind)
  { case RegisterA:
    ... /* Generate code to add the address of base with the */
    ... /* value in the register leaving the result in reg. */
    break;
    default:
    ... /* Generate code to add the address of base with the */
    ... /* value of index leaving the result in register reg. */
    break;
  } /* switch */
  base->ex_kind = RegisterA;
  base->ex_inv = reg;
  break;
} /* switch */
get next element on list;
base->ex_type = typer;
} /* while */
return(base);
} /* arrayref */
```