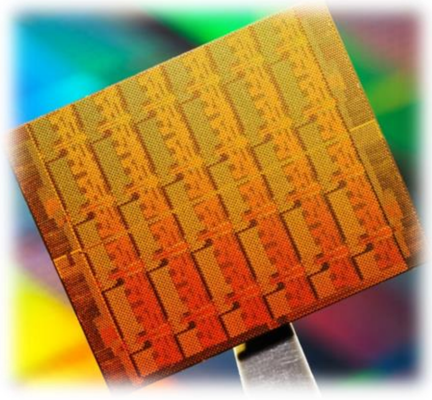
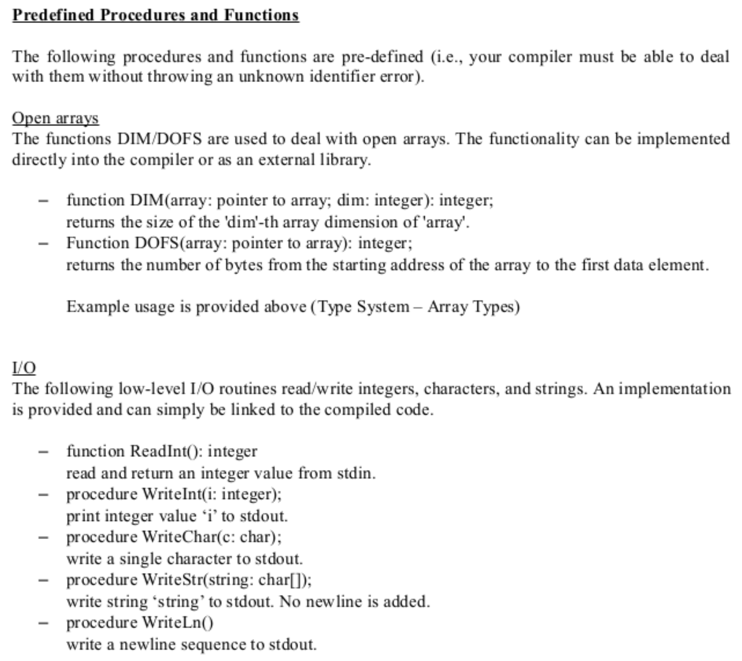
Term Project

Building a Compiler from Scratch



Phase 2 Clarifications

- **Predefined functions and procedures**
  - before starting the parse, add the predefined procedures and functions to the global symbol table (including formal parameters and return type)
  - the project overview slide lists all procedures and functions that are considered predefined on the last page
  - array-related functions
    - DIM(a: ptr to array; dim: integer): integer
    - DOFS(a: ptr to array): integer
  - I/O-related functions
    - ReadInt(): integer
    - WriteInt(i: integer)
    - WriteChar(c: char)
    - WriteStr(str: char[])
    - WriteLn()



Predefined Procedures and Functions

The following procedures and functions are pre-defined (i.e., your compiler must be able to deal with them without throwing an unknown identifier error).

Open arrays
The functions DIM/DOFS are used to deal with open arrays. The functionality can be implemented directly into the compiler or as an external library.

- function DIM(array: pointer to array; dim: integer): integer;
  returns the size of the 'dim'-th array dimension of 'array'.
- Function DOFS(array: pointer to array): integer;
  returns the number of bytes from the starting address of the array to the first data element.

  Example usage is provided above (Type System – Array Types)

I/O
The following low-level I/O routines read/write integers, characters, and strings. An implementation is provided and can simply be linked to the compiled code.

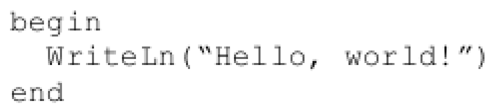
- function ReadInt(): integer
  read and return an integer value from stdin.
- procedure WriteInt(i: integer);
  print integer value 'i' to stdout.
- procedure WriteChar(c: char);
  write a single character to stdout.
- procedure WriteStr(string: char[]);
  write string 'string' to stdout. No newline is added.
- procedure WriteLn()
  write a newline sequence to stdout.

Phase 2 Clarifications

- **Arrays**
  - we only support multi-dimensional arrays
    (even though the printed type in the type manager seems to suggest that arrays are multi-level!)
  - you do not have to support
    - array assignments
    - arrays returned as function values
  - accepting these elements as part of the parsing phase is okay as we will only do type checking in the next phase
  - direct use vs. reference (array vs. pointer to array)
    - array parameters are implicitly passed as references
    - this will require special care when
      - accessing arrays
      - passing them as parameters

Phase 2 Clarifications

- **Strings**
  - also here, refer to the project overview on eTL
  - strings are not a separate type, but rather treated as character arrays, i.e., string = char[]
  - you only have to support immutable strings since array assignments are optional. That means you do not have to support
    - var str: char[20]; … str := “Hello, world!”;
    - you can implement array copy in SnuPL/1 code
  - note: the example in the project overview is incorrect; WriteLn() doesn’t take a parameter



```
begin
  WriteLn("Hello, world!")
end
```

Phase 2 Clarifications

- **AST**
  - we mentioned that you don’t have to modify the AST in this phase.
    This is not entirely correct.
  - to parse SnuPL/-1, the LHS of CAstStatAssign is a CAstConstant, however, for SnuPL/1 this must be of type CAstDesignator.

Phase 2 Clarifications

- **Using the reference scanner**
  - won’t work on Cygwin under Windows
    - use Linux natively or in a VM
  - if the tokens defined in your implementation of the scanner differ from our reference implementation, you may have to modify the SnuPL/-1 parser provided with the handout, but other than that there are no consequences
  - our implementation of CToken has two static methods, escape() and unescape() to escape and unescape strings. If you use the reference scanner, you can use them in your code where necessary, otherwise you will have to implement them yourself (doesn’t have to be static methods in CToken)

Phase 2 Clarifications

- **Reference implementation**
  - our reference implementation already prints some type information…
  - …and unfortunately, some of it is wrong (i.e., if you compile test06.mod, you get lots of <INVALID> types)
  - type checking will be phase 3, so you can delay all type-check related code.
  - in principle, checks that test
    ① equality of the module ID at the beginning/end of the module
    ② equality of the subroutine ID at the beginning/end of the subroutine
    ③ prior declaration of variable uses
    ④ checking whether a variable is actually an array when parsing array dereferencing code

    are semantic checks and not part of the syntax checking phase. You should to (1) and (2) in this phase, (3) and (4) can be delayed.