

# A New Compiler for Parallel Programming

or: having fun building compilers

or maybe: how not to build compilers



**Fred Barnes, Systems Research Group**  
**Computing Laboratory, University of Kent**  
**F.R.M.Barnes@kent.ac.uk**



## Contents

- Motivation (and why a compiler written in C)
- Structure
- Dynamics
- Fun things
- Current and future work

# Motivation

## Motivation

- Current `occam-π` compiler getting old
  - Inmos `occam` compiler first released 1987, acquired by UKC in 1996, open-sourced in 1998, attacked by Fred in 2000
  - now some 148,000 lines of C
  - designed to run in under 2 megabytes of memory
  - not designed to handle the sorts of things we're putting in `occam-π` (e.g. pointers in structures)

## Motivation

- ▶ Current occam- $\pi$  compiler getting old
  - Inmos occam compiler first released 1987, acquired by UKC in 1996, open-sourced in 1998, attacked by Fred in 2000
  - now some 148,000 lines of C
  - designed to run in under 2 megabytes of memory
  - not designed to handle the sorts of things we're putting in occam- $\pi$  (e.g. pointers in structures)
- ▶ Some points:
  - parser in the current compiler is hard-coded
  - occam needs a parser that can look-ahead 2 tokens; occam- $\pi$  requires 3

# Motivation

## Motivation

- ▶ What I wanted in a new compiler (requirements):

## Motivation

- ▶ What I wanted in a new compiler (requirements):
  - a nice compiler framework (developer friendly)

## Motivation

- ▶ What I wanted in a new compiler (requirements):
  - a nice compiler framework (developer friendly)
  - ability to change the language grammar at run-time

## Motivation

- ▶ What I wanted in a new compiler (requirements):
  - a nice compiler framework (developer friendly)
  - ability to change the language grammar at run-time
  - multiple front-ends (languages) and back-ends (code-generators)

## Motivation

- ▶ What I wanted in a new compiler (requirements):
  - a nice compiler framework (developer friendly)
  - ability to change the language grammar at run-time
  - multiple front-ends (languages) and back-ends (code-generators)
  - some separation of language and compiler (partly implied)

## Motivation

- ▶ What I wanted in a new compiler (requirements):
  - a nice compiler framework (developer friendly)
  - ability to change the language grammar at run-time
  - multiple front-ends (languages) and back-ends (code-generators)
  - some separation of language and compiler (partly implied)
  - efficiency!

## Motivation

- ▶ What I wanted in a new compiler (requirements):
  - a nice compiler framework (developer friendly)
  - ability to change the language grammar at run-time
  - multiple front-ends (languages) and back-ends (code-generators)
  - some separation of language and compiler (partly implied)
  - efficiency!
- ▶ Choosing a source language:
  - compilers are, more or less, program transformation tools
  - would make sense to use a functional language
  - also some constraint analysis required – declarative languages ?
  - occam- $\pi$  – bootstrap problem

## Writing it in C

## Writing it in C

➤ I like C ... (and occam- $\pi$ !)

## Writing it in C

- I like C ... (and occam- $\pi$ !)
- Makes for a very portable compiler

## Writing it in C

- I like C ... (and occam- $\pi$ !)
- Makes for a very portable compiler
- Pretty efficient (suprisingly so), fairly easy to debug

## Writing it in C

- I like C ... (and occam- $\pi$ !)
- Makes for a very portable compiler
- Pretty efficient (suprisingly so), fairly easy to debug
- Dynamic loading of compiler code at run-time (extensions)

## Writing it in C

- I like C ... (and occam- $\pi$ !)
- Makes for a very portable compiler
- Pretty efficient (suprisingly so), fairly easy to debug
- Dynamic loading of compiler code at run-time (extensions)
- Can change the behaviour of the compiler dynamically at run-time
  - the way the compiler fits together makes this a bit like “mixins” [1] (though these were aimed primarily at OO languages)
  - I would rather do it manually (not trying to hide things)

## Writing it in C

- I like C ... (and occam- $\pi$ !)
- Makes for a very portable compiler
- Pretty efficient (suprisingly so), fairly easy to debug
- Dynamic loading of compiler code at run-time (extensions)
- Can change the behaviour of the compiler dynamically at run-time
  - the way the compiler fits together makes this a bit like “mixins” [1] (though these were aimed primarily at OO languages)
  - I would rather do it manually (not trying to hide things)
- gcc, gdb and valgrind are getting pretty good these days

## Writing it in C

- I like C ... (and occam- $\pi$ !)
- Makes for a very portable compiler
- Pretty efficient (suprisingly so), fairly easy to debug
- Dynamic loading of compiler code at run-time (extensions)
- Can change the behaviour of the compiler dynamically at run-time
  - the way the compiler fits together makes this a bit like “mixins” [1] (though these were aimed primarily at OO languages)
  - I would rather do it manually (not trying to hide things)
- gcc, gdb and valgrind are getting pretty good these days
- We understand where C goes wrong/bad/ugly more so than for some other languages (I think)

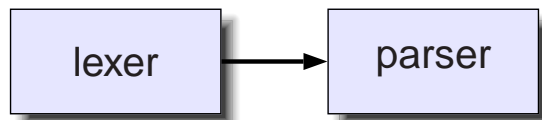
# Structure

## Structure

- ▶ An extensible monolithic multi-pass compiler:

## Structure

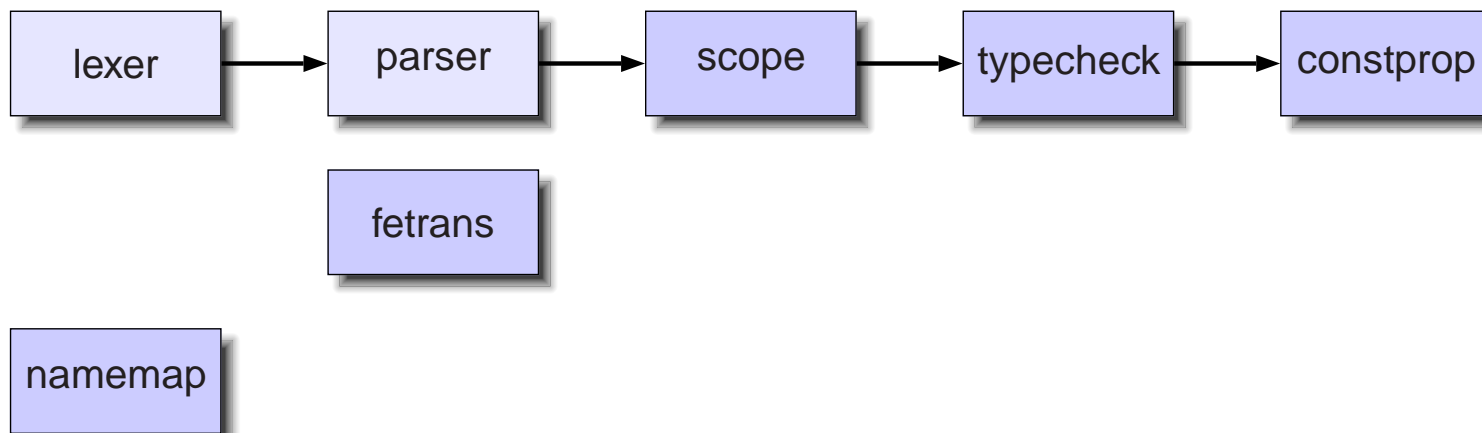
- ▶ An extensible monolithic multi-pass compiler:
  - very language-specific parts



## Structure

► An extensible monolithic multi-pass compiler:

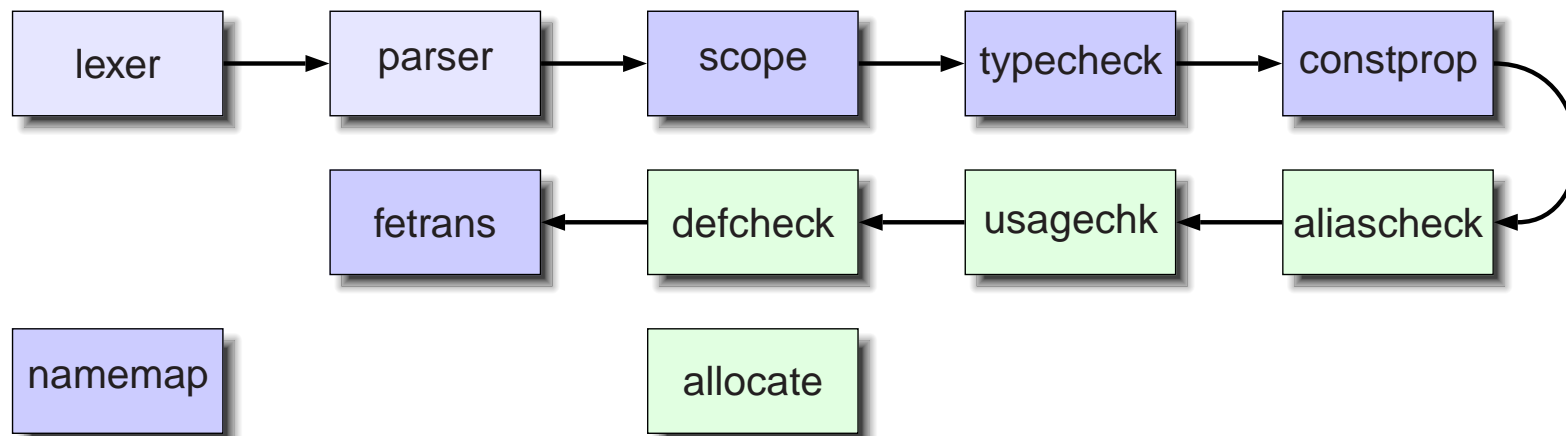
- very language-specific parts
- fairly language-specific parts



## Structure

► An extensible monolithic multi-pass compiler:

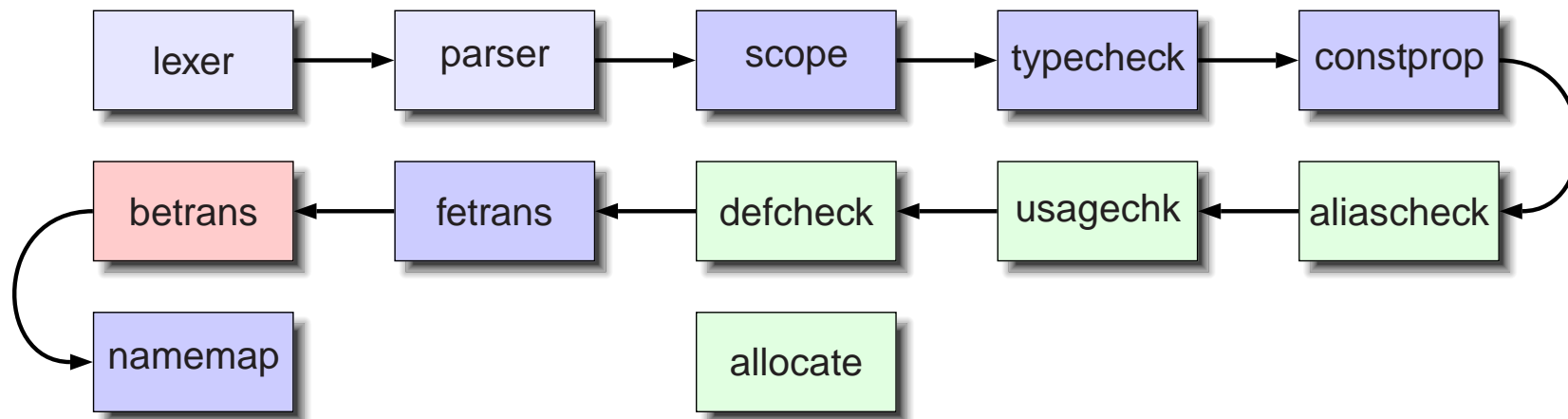
- very language-specific parts
- fairly language-specific parts
- not so language or target specific parts



## Structure

➤ An extensible monolithic multi-pass compiler:

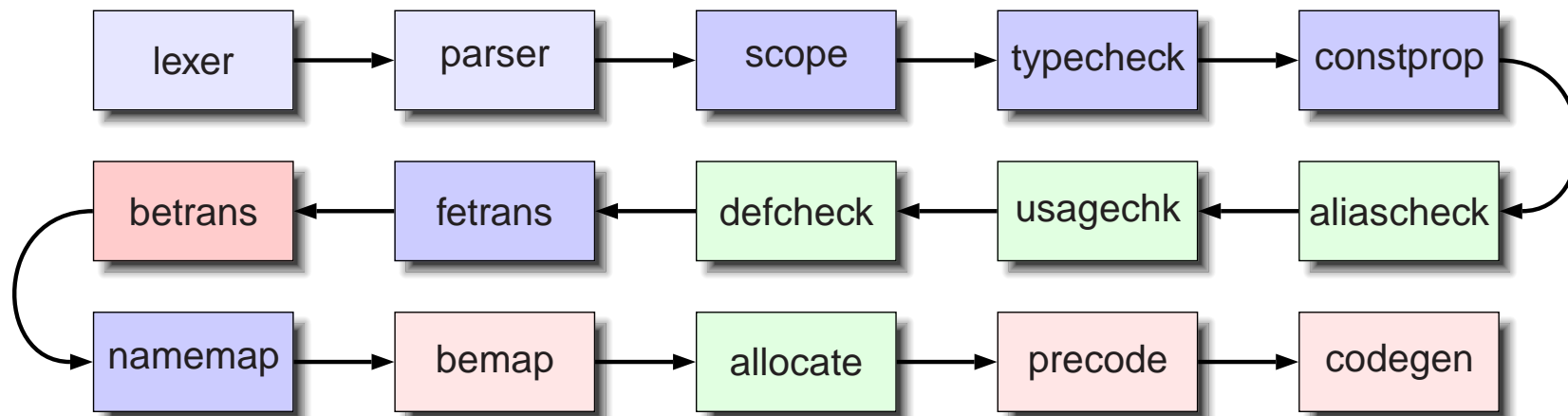
- very language-specific parts
- fairly language-specific parts
- not so language or target specific parts
- fairly target-specific parts



## Structure

► An extensible monolithic multi-pass compiler:

- very language-specific parts
- fairly language-specific parts
- not so language or target specific parts
- fairly target-specific parts
- very target-specific parts

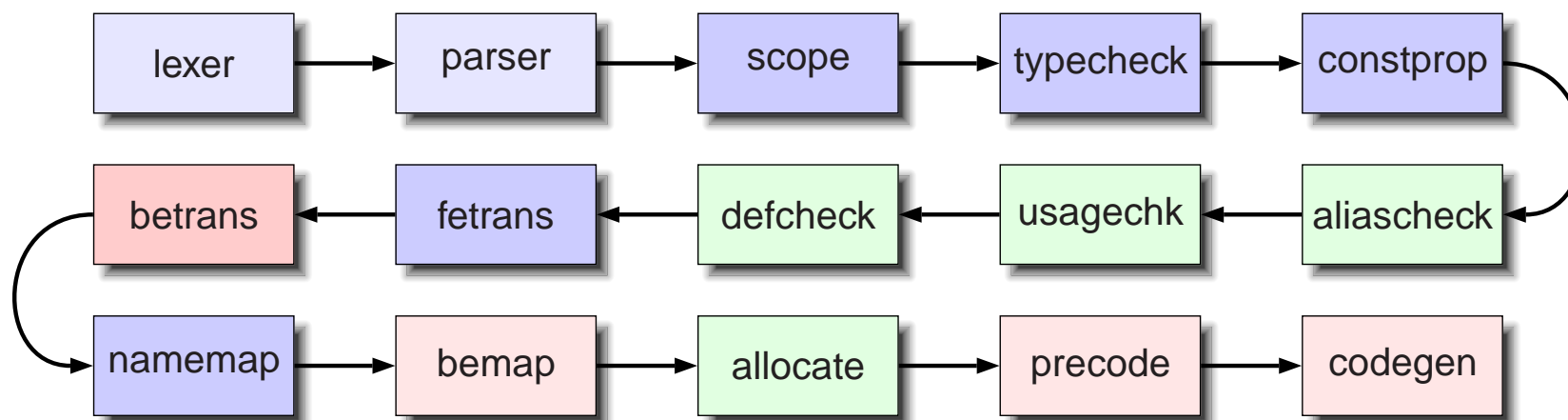


# Structure

➤ An extensible monolithic multi-pass compiler:

- very language-specific parts
- fairly language-specific parts
- not so language or target specific parts
- fairly target-specific parts
- very target-specific parts

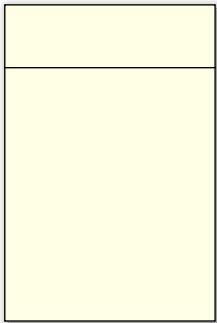
Compiler extensions can insert their own passes into the compiler if needed



# Parse Trees

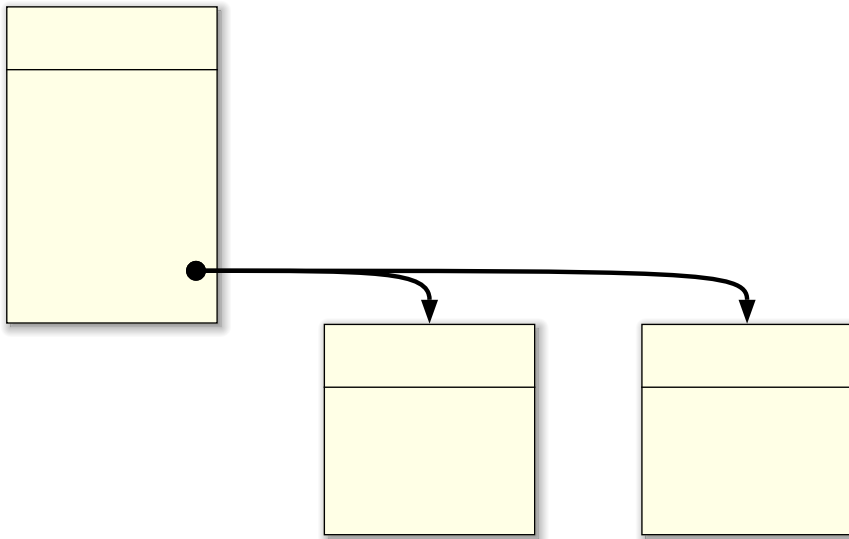
## Parse Trees

- ▶ Parse trees are built from a generic `tnode_t` structure type – not a union



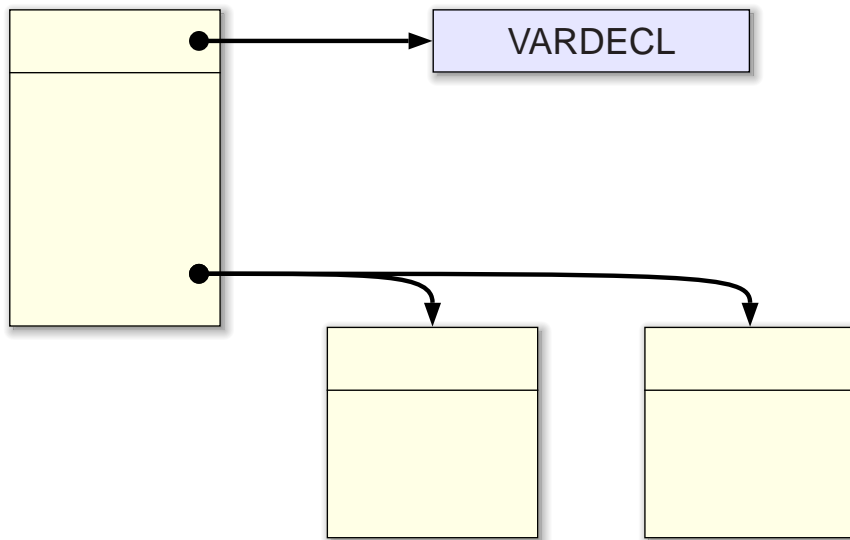
## Parse Trees

- ▶ Parse trees are built from a generic `tnode_t` structure type – not a union
  - tree nodes have subtrees, subnames and hooks



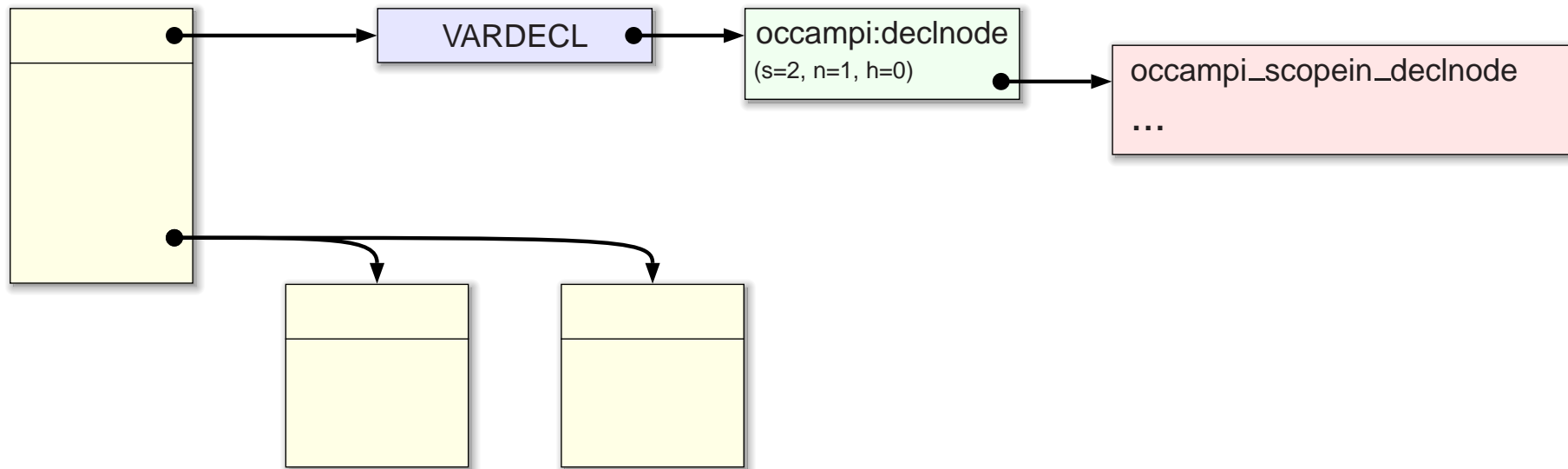
## Parse Trees

- ▶ Parse trees are built from a generic `tnode_t` structure type – not a union
  - tree nodes have subtrees, subnames and hooks
- ▶ Nodes identified with `ntdef_t` tags (mostly for naming)



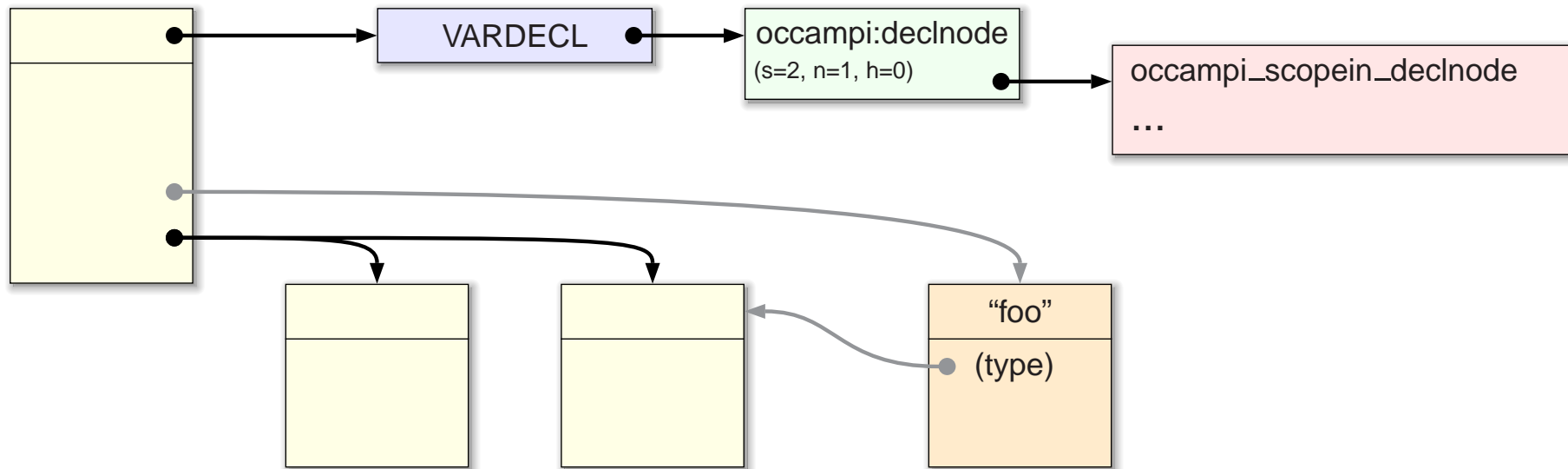
## Parse Trees

- ▶ Parse trees are built from a generic `tnode_t` structure type – not a union
  - tree nodes have subtrees, subnames and hooks
- ▶ Nodes identified with `ntdef_t` tags (mostly for naming)
  - these are linked to `tndef_t` node-type definitions, holding various function pointers used to operate on tree nodes



## Parse Trees

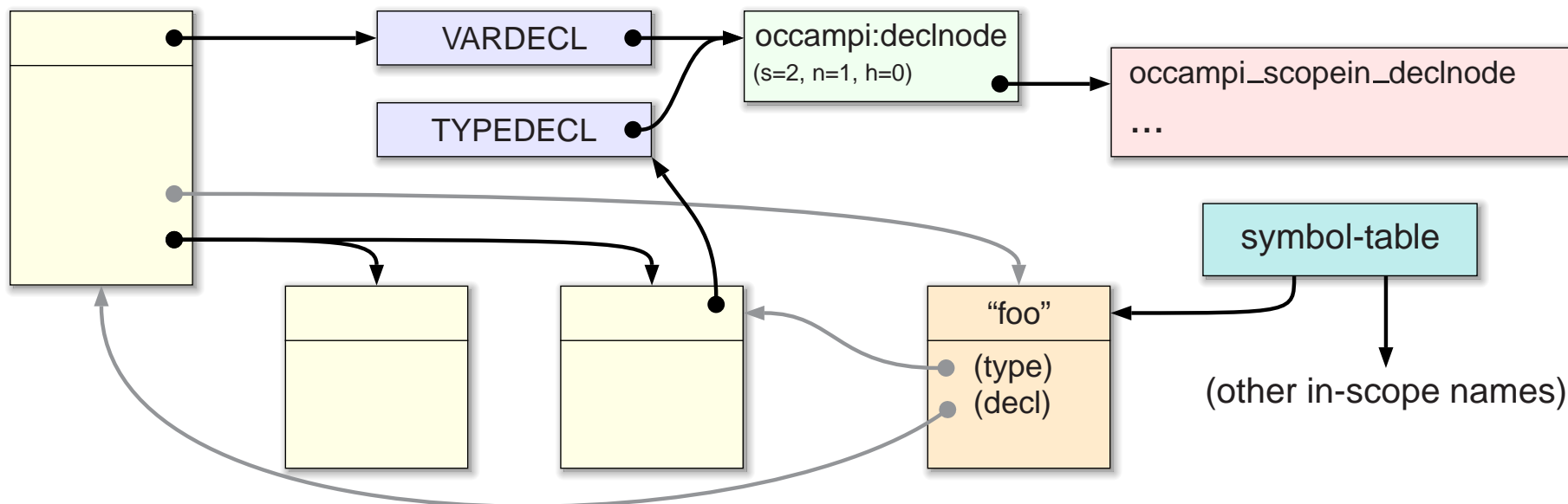
- ▶ Parse trees are built from a generic `tnode_t` structure type – not a union
  - tree nodes have subtrees, subnames and hooks
- ▶ Nodes identified with `ntdef_t` tags (mostly for naming)
  - these are linked to `tndef_t` node-type definitions, holding various function pointers used to operate on tree nodes
- ▶ Names have a special `name_t` type and appear during scoping





## Parse Trees

- Parse trees are built from a generic `tnode_t` structure type – not a union
  - tree nodes have subtrees, subnames and hooks
- Nodes identified with `ntdef_t` tags (mostly for naming)
  - these are linked to `tndef_t` node-type definitions, holding various function pointers used to operate on tree nodes
- Names have a special `name_t` type and appear during scoping



# Building the Compiler

## Building the Compiler

- ▶ When it starts up, the compiler is 'empty' – node types and tags are populated on language initialisation

## Building the Compiler

- ▶ When it starts up, the compiler is 'empty' – node types and tags are populated on language initialisation

```
tndef_t *tnd;  ntdef_t *tag;  
tnd = tnode_newnodetype ("mynode", ...);  
tag = tnode_newnodetag ("MYTAG", tnd, ...);
```

## Building the Compiler

- ▶ When it starts up, the compiler is 'empty' – node types and tags are populated on language initialisation

```
tndef_t *tnd;  ntdef_t *tag;  
tnd = tnode_newnodetype ("mynode", ...);  
tag = tnode_newnodetag ("MYTAG", tnd, ...);
```

- ▶ Followed by grammar constructs used to build the DFAs:

## Building the Compiler

- ▶ When it starts up, the compiler is 'empty' – node types and tags are populated on language initialisation

```
tndef_t *tnd;  ntdef_t *tag;  
tnd = tnode_newnodetype ("mynode", ...);  
tag = tnode_newnodetag ("MYTAG", tnd, ...);
```

- ▶ Followed by grammar constructs used to build the DFAs:

```
dynarray_add (transtbl, dfa_bnftotbl (  
    "mylang:decl ::= mylang:type { mylang:name @@, 1 }"));  
dynarray_add (transtbl, dfa_transtotbl (  
    "mylang:name ::= [ 0 +@Name 1 ] [ 1 {m1:rname} -* ]"));
```

## Building the Compiler

- ▶ When it starts up, the compiler is 'empty' – node types and tags are populated on language initialisation

```
tndef_t *tnd;  ntdef_t *tag;  
tnd = tnode_newnodetype ("mynode", ...);  
tag = tnode_newnodetag ("MYTAG", tnd, ...);
```

- ▶ Followed by grammar constructs used to build the DFAs:

```
dynarray_add (transtbl, dfa_bnftotbl (  
    "mylang:decl ::= mylang:type { mylang:name @@, 1 }"));  
dynarray_add (transtbl, dfa_transtotbl (  
    "mylang:name ::= [ 0 +@Name 1 ] [ 1 {m1:rname} -* ]"));
```

- ▶ Reductions are given in a mini-language:

## Building the Compiler

- ▶ When it starts up, the compiler is 'empty' – node types and tags are populated on language initialisation

```
tndef_t *tnd;  ntdef_t *tag;
tnd = tnode_newnodetype ("mynode", ...);
tag = tnode_newnodetag ("MYTAG", tnd, ...);
```

- ▶ Followed by grammar constructs used to build the DFAs:

```
dynarray_add (transtbl, dfa_bnftotbl (
    "mylang:decl ::= mylang:type { mylang:name @@, 1 }"));
dynarray_add (transtbl, dfa_transtotbl (
    "mylang:name ::= [ 0 +@Name 1 ] [ 1 {ml:rname} -* ]"));
```

- ▶ Reductions are given in a mini-language:

```
parser_register_grule ("ml:rname", .....
    parser_decode_grule ("T+St0XC1R-", mylang_makename, tag));
```

## Building the Compiler

- ▶ When it starts up, the compiler is 'empty' – node types and tags are populated on language initialisation

```
tndef_t *tnd;  ntdef_t *tag;
tnd = tnode_newnodetype ("mynode", ...);
tag = tnode_newnodetag ("MYTAG", tnd, ...);
```

- ▶ Followed by grammar constructs used to build the DFAs:

```
dynarray_add (transtbl, dfa_bnftotbl (
    "mylang:decl ::= mylang:type { mylang:name @@, 1 }"));
dynarray_add (transtbl, dfa_transtotbl (
    "mylang:name ::= [ 0 +@Name 1 ] [ 1 {ml:rname} -* ]"));
```

- ▶ Reductions are given in a mini-language:

```
parser_register_grule ("ml:rname", .....
    parser_decode_grule ("T+St0XC1R-", mylang_makename, tag));
```

- ▶ The generic reducer is a stack machine, able to push/pop tokens/nodes, manipulate the DFA state-machine and call arbitrary functions

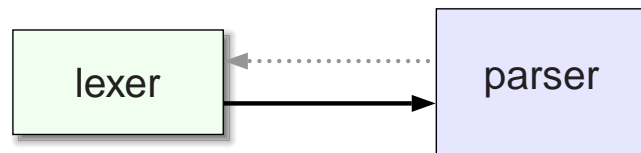
## Parsing Sources

## Parsing Sources

- ▶ The DFA state machine used in this compiler is moderately non-trivial:

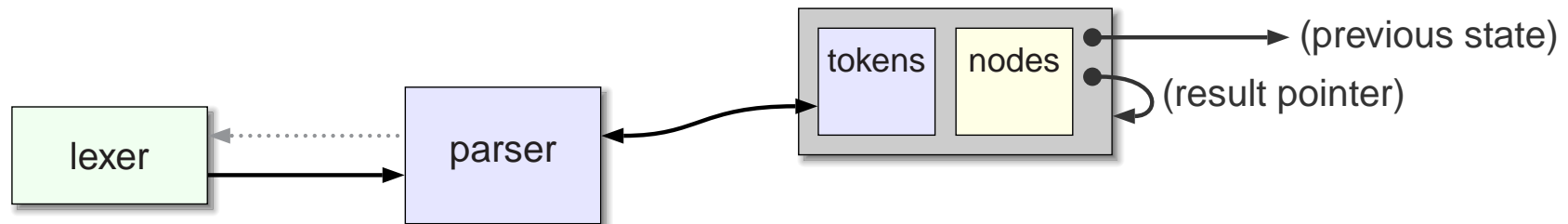
## Parsing Sources

- ▶ The DFA state machine used in this compiler is moderately non-trivial:
  - tokens come out of the lexer, but can be pushed back



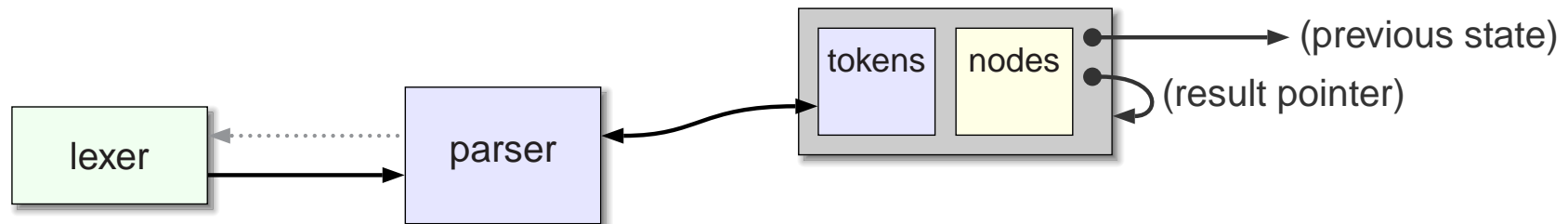
## Parsing Sources

- ▶ The DFA state machine used in this compiler is moderately non-trivial:
  - tokens come out of the lexer, but can be pushed back
  - tokens may be consumed in the parser, or pushed onto a token-stack in the current DFA state



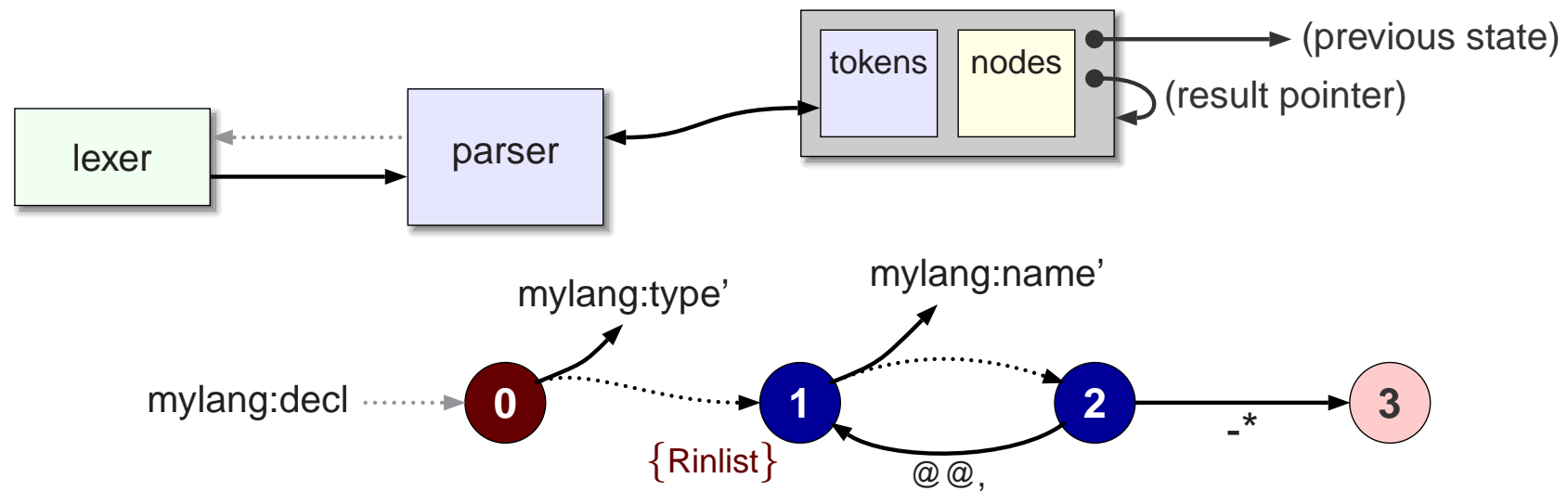
## Parsing Sources

- ▶ The DFA state machine used in this compiler is moderately non-trivial:
  - tokens come out of the lexer, but can be pushed back
  - tokens may be consumed in the parser, or pushed onto a token-stack in the current DFA state
  - DFA state is pushed when branching in the state machine, on return the node result is added to the parent's node-stack



## Parsing Sources

- ▶ The DFA state machine used in this compiler is moderately non-trivial:
  - tokens come out of the lexer, but can be pushed back
  - tokens may be consumed in the parser, or pushed onto a token-stack in the current DFA state
  - DFA state is pushed when branching in the state machine, on return the node result is added to the parent's node-stack



## the Rest of the Compiler

## the Rest of the Compiler

- ▶ Once the parse tree has been built, the rest is mostly tree-transformations

## the Rest of the Compiler

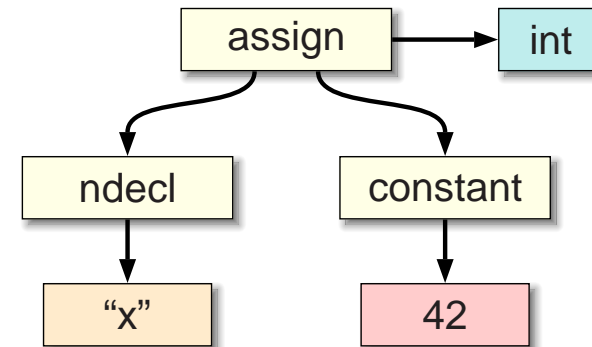
- ▶ Once the parse tree has been built, the rest is mostly tree-transformations
- ▶ Alias, parallel-usage and definedness checks operate on markers left in the tree
  - slightly special case for the **PAR** node itself

## the Rest of the Compiler

- Once the parse tree has been built, the rest is mostly tree-transformations
- Alias, parallel-usage and definedness checks operate on markers left in the tree
  - slightly special case for the **PAR** node itself
- The compiler's `target_t` structure defines various back-end specific nodes

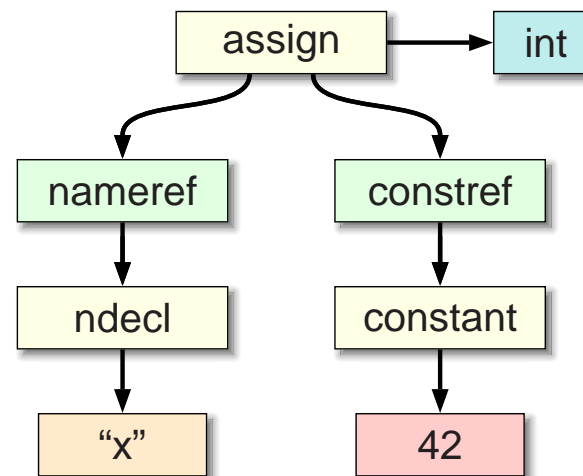
## the Rest of the Compiler

- Once the parse tree has been built, the rest is mostly tree-transformations
- Alias, parallel-usage and definedness checks operate on markers left in the tree
  - slightly special case for the **PAR** node itself
- The compiler's `target_t` structure defines various back-end specific nodes
- Name-map pass inserts these



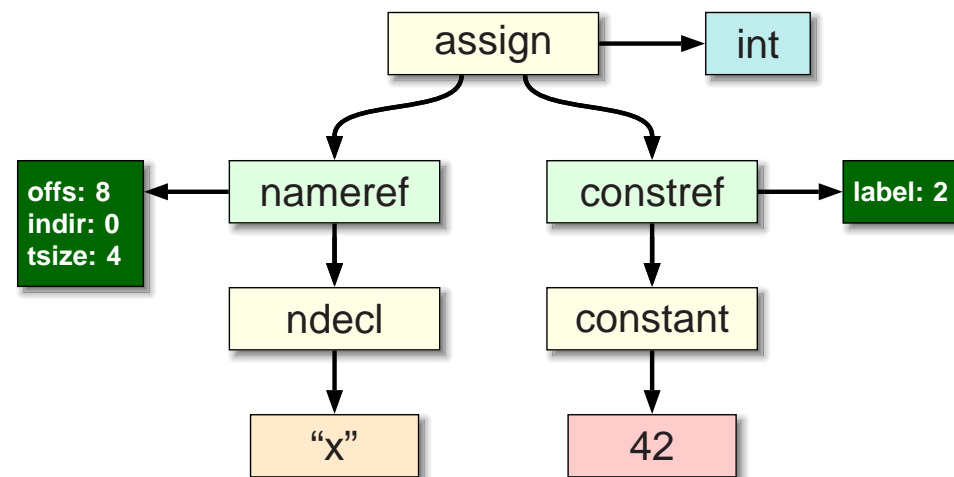
## the Rest of the Compiler

- Once the parse tree has been built, the rest is mostly tree-transformations
- Alias, parallel-usage and definedness checks operate on markers left in the tree
  - slightly special case for the **PAR** node itself
- The compiler's `target_t` structure defines various back-end specific nodes
- Name-map pass inserts these



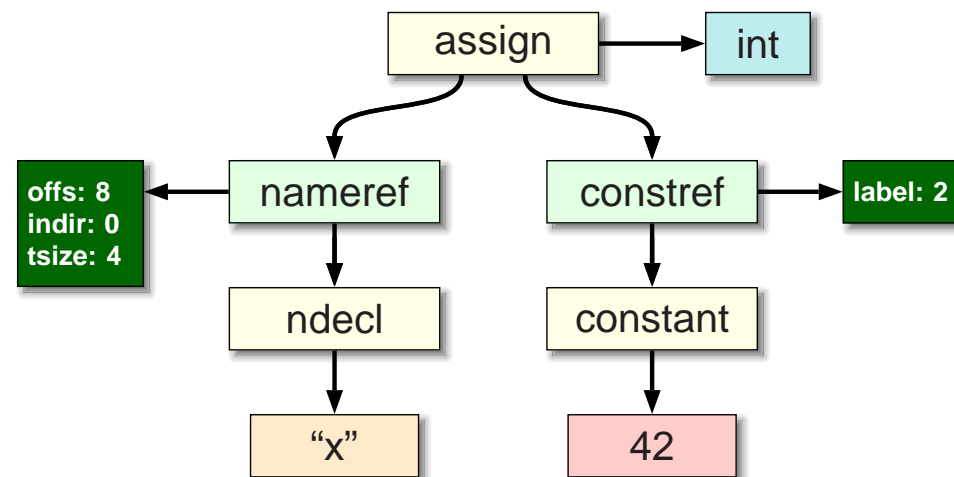
## the Rest of the Compiler

- Once the parse tree has been built, the rest is mostly tree-transformations
- Alias, parallel-usage and definedness checks operate on markers left in the tree
  - slightly special case for the **PAR** node itself
- The compiler's `target_t` structure defines various back-end specific nodes
- Name-map pass inserts these
- Allocate pass determines memory requirements and allocates offsets



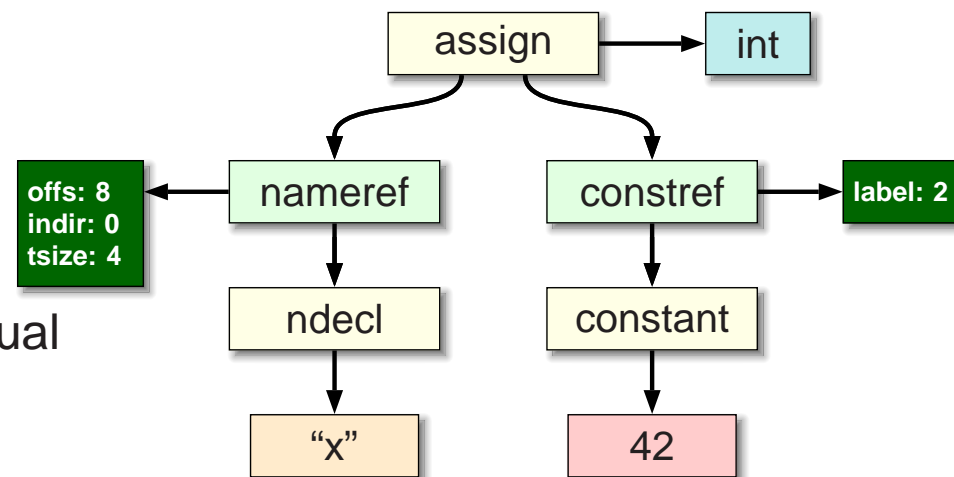
## the Rest of the Compiler

- Once the parse tree has been built, the rest is mostly tree-transformations
- Alias, parallel-usage and definedness checks operate on markers left in the tree
  - slightly special case for the **PAR** node itself
- The compiler's `target_t` structure defines various back-end specific nodes
- Name-map pass inserts these
- Allocate pass determines memory requirements and allocates offsets
- Code-generation pass then deals mostly with back-end nodes only



## the Rest of the Compiler

- Once the parse tree has been built, the rest is mostly tree-transformations
- Alias, parallel-usage and definedness checks operate on markers left in the tree
  - slightly special case for the **PAR** node itself
- The compiler's `target_t` structure defines various back-end specific nodes
- Name-map pass inserts these
- Allocate pass determines memory requirements and allocates offsets
- Code-generation pass then deals mostly with back-end nodes only
- Default target is a T800 flavoured virtual transputer (for KRoC)



## Multiple Languages

## Multiple Languages

- ▶ The compiler currently supports two languages:
  - occam- $\pi$ , about 40% complete; MCSP, about 50% complete

## Multiple Languages

- ▶ The compiler currently supports two languages:
  - occam- $\pi$ , about 40% complete; MCSP, about 50% complete
- ▶ The MCSP language (for compiling CSP-ish scripts) is much simpler

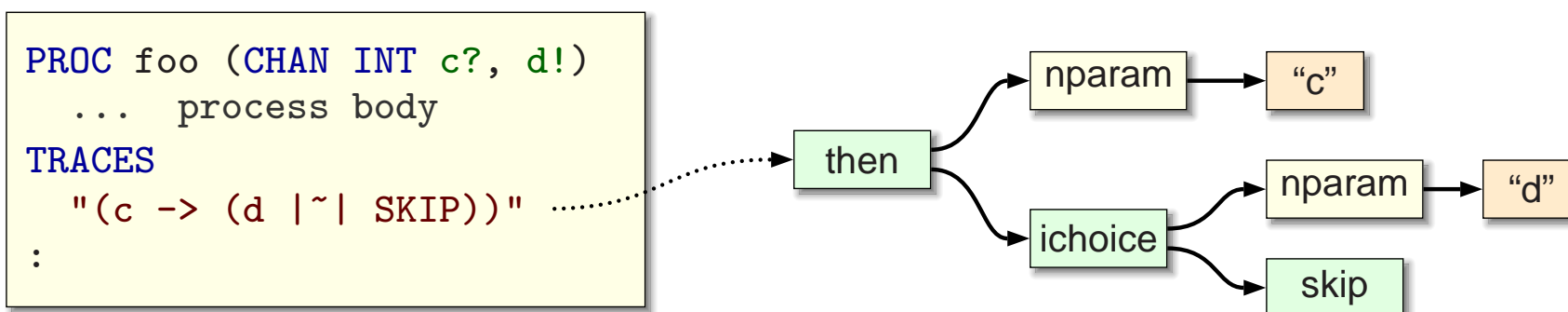
## Multiple Languages

- ▶ The compiler currently supports two languages:
  - occam- $\pi$ , about 40% complete; MCSP, about 50% complete
- ▶ The MCSP language (for compiling CSP-ish scripts) is much simpler
- ▶ The MCSP parser can be invoked from within occam- $\pi$ , e.g.:

```
PROC foo (CHAN INT c?, d!)  
  ... process body  
TRACES  
  "(c -> (d |~| SKIP))"  
:
```

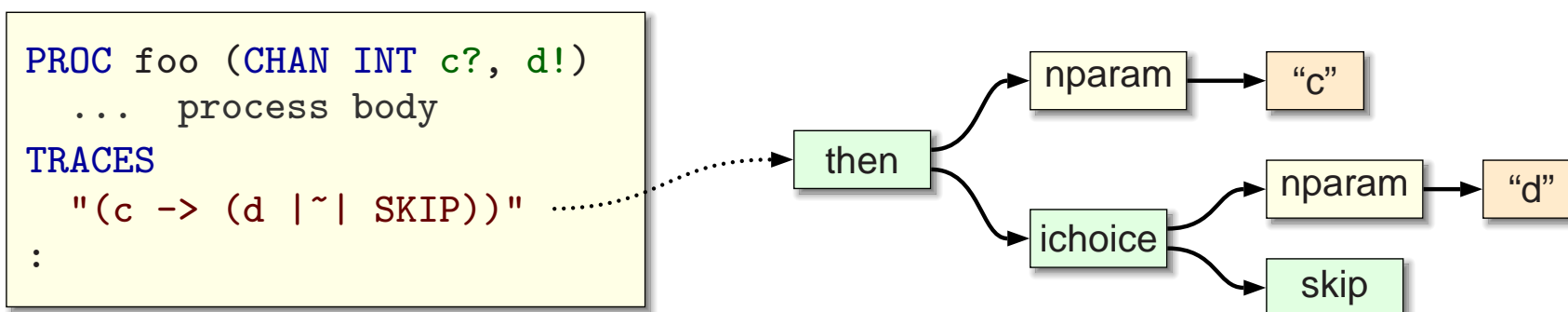
## Multiple Languages

- ▶ The compiler currently supports two languages:
  - occam- $\pi$ , about 40% complete; MCSP, about 50% complete
- ▶ The MCSP language (for compiling CSP-ish scripts) is much simpler
- ▶ The MCSP parser can be invoked from within occam- $\pi$ , e.g.:



## Multiple Languages

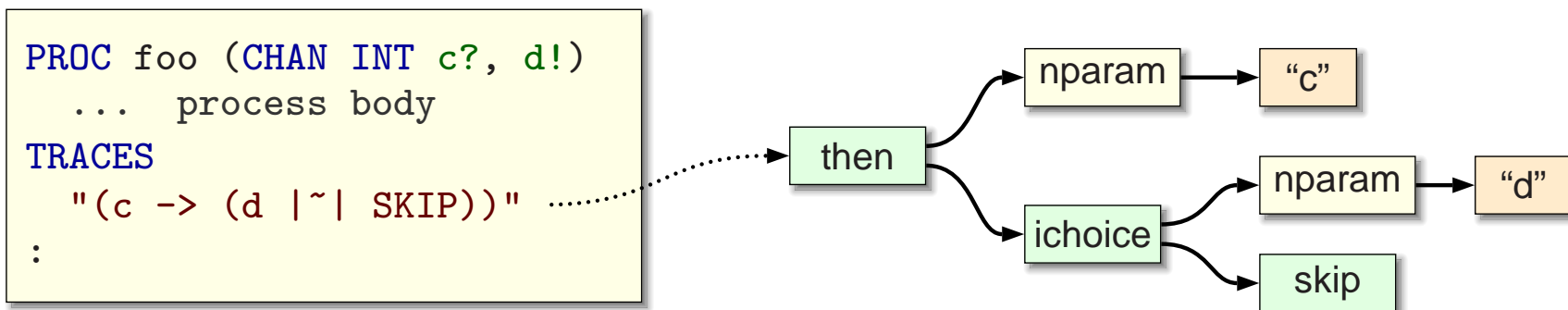
- The compiler currently supports two languages:
  - occam- $\pi$ , about 40% complete; MCSP, about 50% complete
- The MCSP language (for compiling CSP-ish scripts) is much simpler
- The MCSP parser can be invoked from within occam- $\pi$ , e.g.:



- Not fully implemented yet, but will be used to check process conformance
  - also intending it to work in reverse (compiler generating behaviours)

## Multiple Languages

- The compiler currently supports two languages:
  - occam- $\pi$ , about 40% complete; MCSP, about 50% complete
- The MCSP language (for compiling CSP-ish scripts) is much simpler
- The MCSP parser can be invoked from within occam- $\pi$ , e.g.:



- Not fully implemented yet, but will be used to check process conformance
  - also intending it to work in reverse (compiler generating behaviours)
- Also some parts of a BASIC front-end in place (continuation as MSc project)

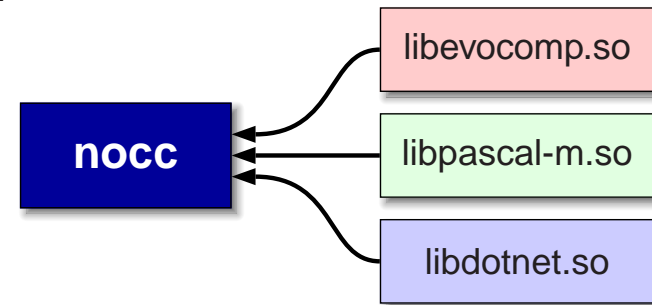
## Extending the Compiler

## Extending the Compiler

- Various mechanisms in place (or under construction) to allow changes to the compiler at run-time:

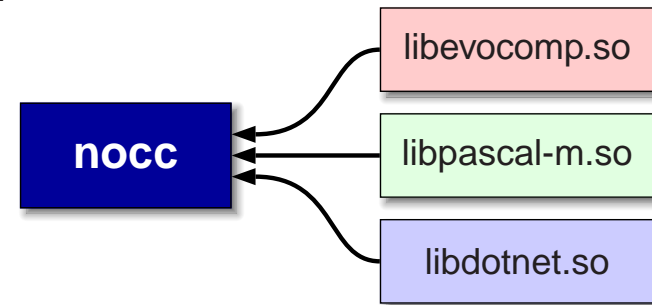
## Extending the Compiler

- Various mechanisms in place (or under construction) to allow changes to the compiler at run-time:
- Compiler extensions – shared objects loaded into the compiler at run-time
  - could provide new languages, compiler features or code-generators



## Extending the Compiler

- Various mechanisms in place (or under construction) to allow changes to the compiler at run-time:
- Compiler extensions – shared objects loaded into the compiler at run-time
  - could provide new languages, compiler features or code-generators
- Language extensions – for experimenting with language features

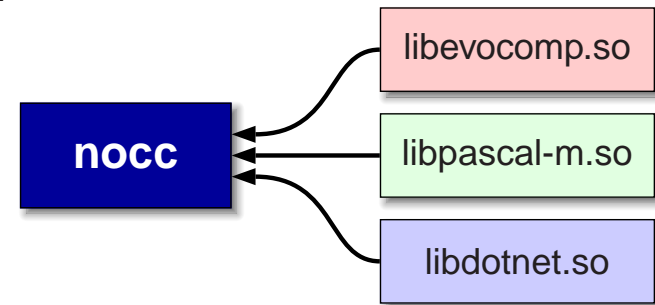


## Extending the Compiler

➤ Various mechanisms in place (or under construction) to allow changes to the compiler at run-time:

➤ Compiler extensions – shared objects loaded into the compiler at run-time

- could provide new languages, compiler features or code-generators



➤ Language extensions – for experimenting with language features

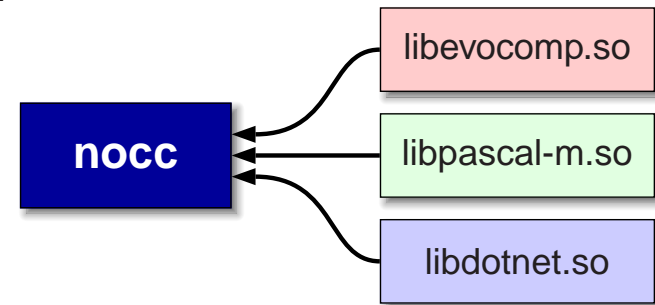
➤ Compiler add-ons – descriptions loaded into the compiler to add trivial features

## Extending the Compiler

➤ Various mechanisms in place (or under construction) to allow changes to the compiler at run-time:

➤ Compiler extensions – shared objects loaded into the compiler at run-time

- could provide new languages, compiler features or code-generators



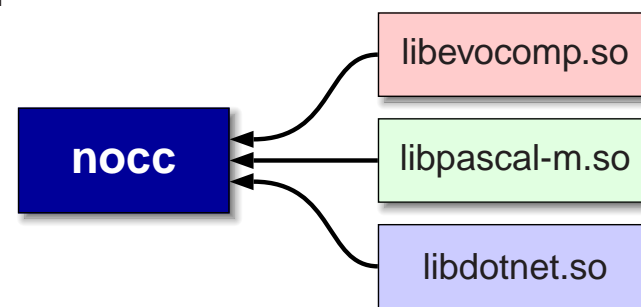
➤ Language extensions – for experimenting with language features

➤ Compiler add-ons – descriptions loaded into the compiler to add trivial features

```
INITIAL INT x IS 14:  
P
```

## Extending the Compiler

- Various mechanisms in place (or under construction) to allow changes to the compiler at run-time:
- Compiler extensions – shared objects loaded into the compiler at run-time
  - could provide new languages, compiler features or code-generators
- Language extensions – for experimenting with language features
- Compiler add-ons – descriptions loaded into the compiler to add trivial features

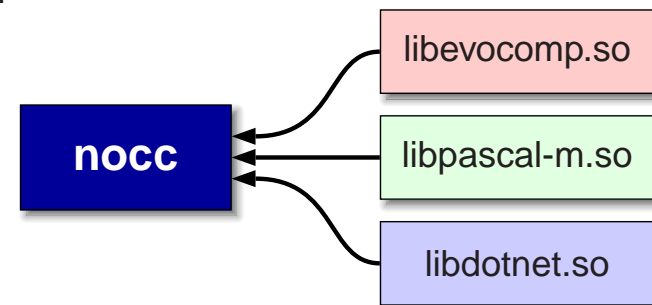


```
INITIAL INT x IS 14:
P
```

```
INT $tmp:
SEQ
  $tmp := 14
INT x IS $tmp:
P
```

## Extending the Compiler

- Various mechanisms in place (or under construction) to allow changes to the compiler at run-time:
- Compiler extensions – shared objects loaded into the compiler at run-time
  - could provide new languages, compiler features or code-generators
- Language extensions – for experimenting with language features
- Compiler add-ons – descriptions loaded into the compiler to add trivial features



```
INITIAL INT x IS 14:
```

```
P
```

```
<match id="idecl" in="top(occampi:decl)">
  <expr>@INITIAL t:type n:name @IS e:expr @@:
        Newline p:process</expr>
  <replace>(vardecl tmp:tempname #t _ (seq
    (assign #tmp #e #t)
    (abbr #n #t #tmp #p)))</replace>
</match>
```

```
INT $tmp:
SEQ
  $tmp := 14
  INT x IS $tmp:
P
```

## Extending Languages

## Extending Languages

- ▶ The KRoC distribution includes various user-defined synchronisation types
  - e.g. 'CREW', 'BUCKET'

## Extending Languages

- ▶ The KRoC distribution includes various user-defined synchronisation types
  - e.g. 'CREW', 'BUCKET'
- ▶ Lacking a way to tell the compiler how to initialise and check these

## Extending Languages

- ▶ The KRoC distribution includes various user-defined synchronisation types
  - e.g. 'CREW', 'BUCKET'
- ▶ Lacking a way to tell the compiler how to initialise and check these

```
DATA TYPE BUCKET
  RECORD
    ... fields
  :
PROC init.bucket (BUCKET b)
  ... process body
  :
PROC wait.bucket (BUCKET b)
  ... process body
  :
PROC empty.bucket (BUCKET b)
  ... process body
  :
```

## Extending Languages

- ▶ The KRoC distribution includes various user-defined synchronisation types
  - e.g. 'CREW', 'BUCKET'
- ▶ Lacking a way to tell the compiler how to initialise and check these

```

DATA TYPE BUCKET
  RECORD
    ... fields
  :
PROC init.bucket (BUCKET b)
  ... process body
  :
PROC wait.bucket (BUCKET b)
  ... process body
  :
PROC empty.bucket (BUCKET b)
  ... process body
  :

```

```

EXTENDED DATA TYPE BUCKET
  RECORD
    ... fields
  PROC .init (BUCKET b)
    ... process body
  :
  ACTION WAIT (BUCKET b)
    ... process body
  :
  ... other PROCs/ACTIONS
  PAR READ, WRITE, WAIT
  :

```

## Extending Languages

- The KRoC distribution includes various user-defined synchronisation types
  - e.g. 'CREW', 'BUCKET'
- Lacking a way to tell the compiler how to initialise and check these

```

DATA TYPE BUCKET
  RECORD
    ... fields
  :
  PROC init.bucket (BUCKET b)
    ... process body
  :
  PROC wait.bucket (BUCKET b)
    ... process body
  :
  PROC empty.bucket (BUCKET b)
    ... process body
  :

```

```

EXTENDED DATA TYPE BUCKET
  RECORD
    ... fields
  PROC .init (BUCKET b)
    ... process body
  :
  ACTION WAIT (BUCKET b)
    ... process body
  :
  ... other PROCs/ACTIONS
  PAR READ, WRITE, WAIT
  :

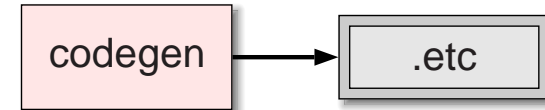
```

- Other types more complex, like those that 'protect' other data (e.g. 'CREW')
  - might want to resort to loadable modules, further investigation needed

## Compiler Output

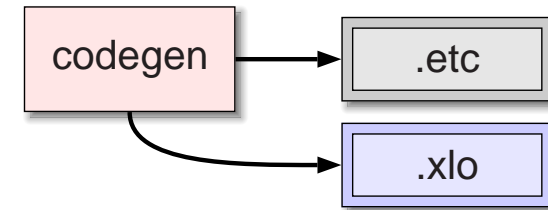
## Compiler Output

- ▶ Primary output is for the selected target (t800/kroc)
  - translated and linked through KRoC as normal



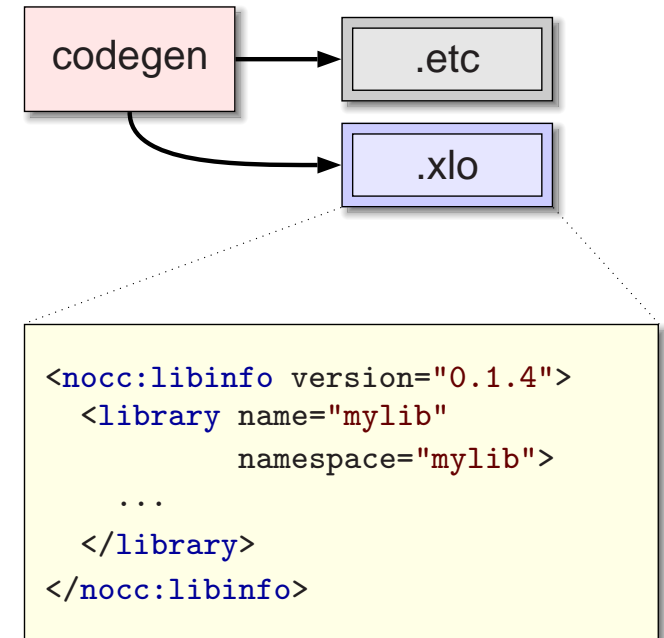
## Compiler Output

- ▶ Primary output is for the selected target (t800/kroc)
  - translated and linked through KRoC as normal
- ▶ Also output, largely for separate compilation and library generation, are `.xlo` files
  - triggered from internal 'library' tree-nodes



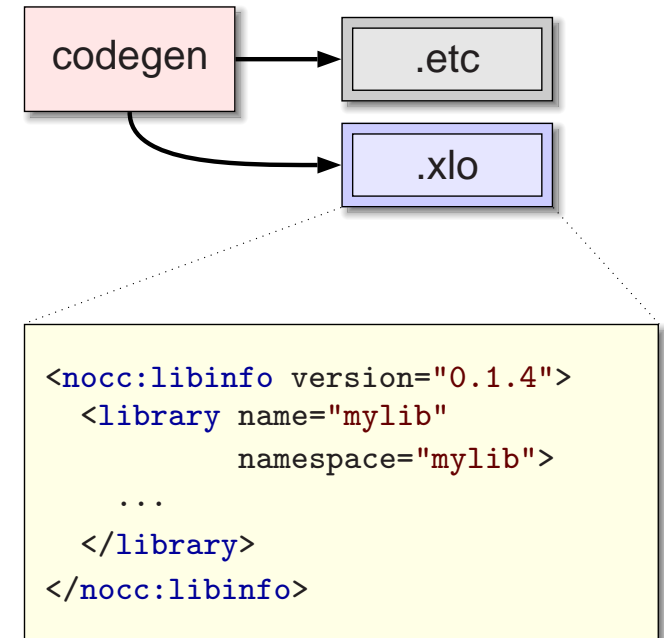
## Compiler Output

- ▶ Primary output is for the selected target (t800/kroc)
  - translated and linked through KRoC as normal
- ▶ Also output, largely for separate compilation and library generation, are `.xlo` files
  - triggered from internal 'library' tree-nodes
- ▶ Contains information about entry-points and descriptors; XML formatting
  - descriptors are re-read in the parser



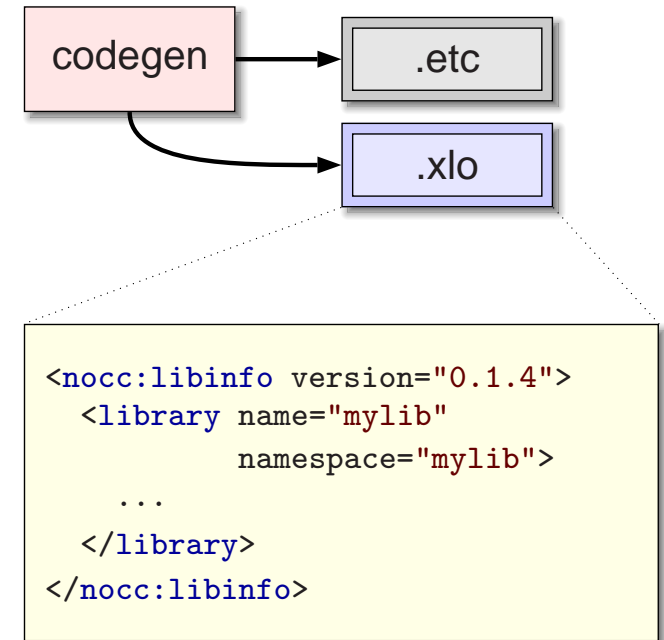
## Compiler Output

- ▶ Primary output is for the selected target (t800/kroc)
  - translated and linked through KRoC as normal
- ▶ Also output, largely for separate compilation and library generation, are `.xlo` files
  - triggered from internal 'library' tree-nodes
- ▶ Contains information about entry-points and descriptors; XML formatting
  - descriptors are re-read in the parser
- ▶ In the long-term, will permit linking of routines from different languages



## Compiler Output

- ▶ Primary output is for the selected target (t800/kroc)
  - translated and linked through KRoC as normal
- ▶ Also output, largely for separate compilation and library generation, are `.xlo` files
  - triggered from internal 'library' tree-nodes
- ▶ Contains information about entry-points and descriptors; XML formatting
  - descriptors are re-read in the parser
- ▶ In the long-term, will permit linking of routines from different languages
- ▶ If desired, compiler will also produce a signed hash for the generated code
  - this should take care of trust issues with dynamic code loading



# Namespaces

## Namespaces

- ▶ Long been a feature that we have wanted in `occam-π`

## Namespaces

- ▶ Long been a feature that we have wanted in `occam-π`
- ▶ Files that contribute code to libraries specify a default namespace

```
#LIBRARY "mylib"  
NATIVELIB "libmylib.so"  
NAMESPACE "mylib"
```

## Namespaces

- ▶ Long been a feature that we have wanted in `occam-π`
- ▶ Files that contribute code to libraries specify a default namespace
- ▶ Code that `#USE`'s a library can do so in a different namespace
- ▶ Also works for ordinary separately compiled code (that have blank namespaces)

```
#LIBRARY "mylib"  
  NATIVELIB "libmylib.so"  
  NAMESPACE "mylib"
```

```
#USE "mylib" AS "out"  
...  
out.foo (...)
```

## Namespaces

- ▶ Long been a feature that we have wanted in `occam-π`
- ▶ Files that contribute code to libraries specify a default namespace
- ▶ Code that `#USE`'s a library can do so in a different namespace
- ▶ Also works for ordinary separately compiled code (that have blank namespaces)
- ▶ Dot `.` is the namespace separator, but is also allowed to be part of a name

```
#LIBRARY "mylib"  
  NATIVELIB "libmylib.so"  
  NAMESPACE "mylib"
```

```
#USE "mylib" AS "out"  
...  
out.foo (...)
```

## Compiling CSP

## Compiling CSP

- Mostly for fun, but has some relevance to the TUNA project [2]

## Compiling CSP

- Mostly for fun, but has some relevance to the TUNA project [2]

```
# this is an MCSP script

FOO (v) ::= @X.(v -> w -> v -> X)
BAR (x) ::= @Y.((x -> ((y -> SKIP) [] (x -> y -> SKIP))); Y)

SYSTEM ::= (FOO (a) || BAR (a)) \ {a}
```

## Compiling CSP

- Mostly for fun, but has some relevance to the TUNA project [2]

```
# this is an MCSP script

FOO (v) ::= @X.(v -> w -> v -> X)
BAR (x) ::= @Y.((x -> ((y -> SKIP) [] (x -> y -> SKIP))); Y)

SYSTEM ::= (FOO (a) || BAR (a)) \ {a}
```

- Uses new multiway-synchronisation support in the KRoC run-time system

## Compiling CSP

- Mostly for fun, but has some relevance to the TUNA project [2]

```
# this is an MCSP script

FOO (v) ::= @X.(v -> w -> v -> X)
BAR (x) ::= @Y.((x -> ((y -> SKIP) [] (x -> y -> SKIP))); Y)

SYSTEM ::= (FOO (a) || BAR (a)) \ {a}
```

- Uses new multiway-synchronisation support in the KRoC run-time system
- Compiler adds parameters for free-events, whole 'SYSTEM' is run in parallel with an 'environment' process
  - offers to synchronise on all events, prints them to the screen when selected

## Compiling CSP

- Mostly for fun, but has some relevance to the TUNA project [2]

```
# this is an MCSP script

FOO (v) ::= @X.(v -> w -> v -> X)
BAR (x) ::= @Y.((x -> ((y -> SKIP) [] (x -> y -> SKIP))); Y)

SYSTEM ::= (FOO (a) || BAR (a)) \ {a}
```

- Uses new multiway-synchronisation support in the KRoC run-time system
- Compiler adds parameters for free-events, whole 'SYSTEM' is run in parallel with an 'environment' process
  - offers to synchronise on all events, prints them to the screen when selected
- MCSP parts of the compiler account for about 5,000 lines of code
  - and not expected to get much bigger :-)

## Current and Future Work

## Current and Future Work

- ▶ Parser for `occam- $\pi$`  is about 50% complete
  - still lacking support for some language features, most of `occam` supported

## Current and Future Work

- ▶ Parser for occam- $\pi$  is about 50% complete
  - still lacking support for some language features, most of occam supported
- ▶ Code-generator for ETC (KRoC route) is mostly complete

## Current and Future Work

- ▶ Parser for  $\text{occam-}\pi$  is about 50% complete
  - still lacking support for some language features, most of  $\text{occam}$  supported
- ▶ Code-generator for ETC (KRoC route) is mostly complete
- ▶ Missing are alias, definedness and some parallel-usage checks
  - need to do some non-trivial constraint analysis for  $\text{occam-}\pi$
  - may end up being a bit prolog-ish in places

## Current and Future Work

- ▶ Parser for  $\text{occam-}\pi$  is about 50% complete
  - still lacking support for some language features, most of  $\text{occam}$  supported
- ▶ Code-generator for ETC (KRoC route) is mostly complete
- ▶ Missing are alias, definedness and some parallel-usage checks
  - need to do some non-trivial constraint analysis for  $\text{occam-}\pi$
  - may end up being a bit prolog-ish in places
- ▶ Useful would be some garbage-collection (for tree-nodes)

## Current and Future Work

- Parser for  $\text{occam-}\pi$  is about 50% complete
  - still lacking support for some language features, most of  $\text{occam}$  supported
- Code-generator for ETC (KRoC route) is mostly complete
- Missing are alias, definedness and some parallel-usage checks
  - need to do some non-trivial constraint analysis for  $\text{occam-}\pi$
  - may end up being a bit prolog-ish in places
- Useful would be some garbage-collection (for tree-nodes)
- Generic routines for breaking/assembling tree nodes in-progress
  - with functional-style mini-programs

## Current and Future Work

- Parser for  $\text{occam-}\pi$  is about 50% complete
  - still lacking support for some language features, most of  $\text{occam}$  supported
- Code-generator for ETC (KRoC route) is mostly complete
- Missing are alias, definedness and some parallel-usage checks
  - need to do some non-trivial constraint analysis for  $\text{occam-}\pi$
  - may end up being a bit prolog-ish in places
- Useful would be some garbage-collection (for tree-nodes)
- Generic routines for breaking/assembling tree nodes in-progress
  - with functional-style mini-programs
- Whole compiler checks in at just less than 50,000 lines of C
  - and works!

## Download and documentation

<http://www.cs.kent.ac.uk/projects/ofa/nocc/>

## References

- [1] Michael Van Hilst and David Notkin. Using Role Components to Implement Collaboration-Based Designs. In *Proceedings of OOPSLA'96, San Jose, USA*, pages 359–369. ACM Press, New York, USA, 1996.
- [2] S. Stepney, P.H. Welch, F.A.C. Pollack, J.C.P. Woodcock, S. Schneider, H.E. Treharne, and A.L.C. Cavalcanti. TUNA: Theory underpinning nanotech assemblers (feasibility study), January 2005. EPSRC grant EP/C516966/1. Available from:  
<http://www.cs.york.ac.uk/nature/tuna/index.htm>.