Compilation of
Tiger to MIPS

Min Liu
Andrew Magill
Table of Contents

1 Introduction
2 Status
   2.1 Things That Work
      2.1.1 Name and Type Analysis
      2.1.2 SPIM
   2.2 Things That Don’t Work Yet
3 Syntactic Analysis
4 Source Tree Construction
   4.1 Abstract Grammar
      4.1.1 Program
      4.1.2 Expressions
         4.1.2.1 Sequence
         4.1.2.2 Literals
         4.1.2.3 Procedure Calls
         4.1.2.4 Arithmetic
         4.1.2.5 Variables
         4.1.2.6 Assignment
         4.1.2.7 Flow Control
         4.1.2.8 Let
      4.1.3 Declarations
         4.1.3.1 Variables
         4.1.3.2 Types
         4.1.3.3 Functions
      4.1.4 Identifiers
   4.2 Concrete Grammar
   4.3 Concrete to abstract mapping
5 Name Analysis
   5.1 Modules
   5.2 Specification
      5.2.1 Name Checking
      5.2.2 Functions
      5.2.3 For
      5.2.4 Let
      5.2.5 Types
6 Type Analysis
   6.1 Modules
   6.2 Inherited Symbols
   6.3 Specification
      6.3.1 Literals
      6.3.2 Operators
      6.3.3 Functions
      6.3.4 Arrays
      6.3.5 Variables
      6.3.6 Expression Sequence
      6.3.7 Type Declaration
      6.3.8 Flow Control
7 Predefined Language Elements
   7.1 Modules
   7.2 Specification
      7.2.1 Types
         7.2.1.1 Identifiers
7.2.1.2 Keys

7.2.2 Operators
  7.2.2.1 Indications
  7.2.2.2 Signatures
  7.2.2.3 Keys

7.2.3 Functions
  7.2.3.1 Indications
  7.2.3.2 Signatures

8 spim
1 Introduction

The goal of this project to perform source-to-source translation (i.e. compilation) from the Tiger programming language into SPIM assembly code.

The Tiger language was introduced as an example in Andrew Appel’s Modern Compiler Implementation book. There is a partial definition presented as an appendix of his book, and since it has been eliminated from future editions, a complete definition has since been posted online.

SPIM is a simulated assembly language written for the RISC-based MIPS microprocessor architecture. There are simulators available for Windows and Linux systems.

This project is specified by the following files, all contained in the FunnelWeb file this documentation was generated from. For the sake of working on the project in fairly independent stages, we broke up the project into five parts: source tree construction, name analysis, type analysis, predefined language elements, and code generation.

\[ \text{tiger.specs} \equiv \text{tiger.lido} \]
\[ \text{name.lido} \]
\[ \text{type.lido} \]
\[ \text{tiger.con} \]
\[ \text{tiger.gla} \]
\[ \text{tiger.pdl} \]
\[ \text{tiger.map} \]
\[ \text{NestedCcomment.c} \]
\[ \text{TigerChStr.c} \]
\[ \text{myfac.h} \]
\[ \text{myfac.head} \]

\text{Spim\%Spim.specs}

\begin{itemize}
  \item \textit{Name Analysis Modules}[23]
  \item \textit{Type Analysis Modules}[30]
  \item \textit{Predefinition Modules}[43]
\end{itemize}

This macro is attached to a product file.
2 Status

2.1 Things That Work

2.1.1 Name and Type Analysis

This is a subset of Tiger without records and nil, and variable declaration without specifying type. And arrays can only be 1 dimensional. In Tiger specification, function and type declarations can be recursive or mutually recursive. However, each type recursion cycle must pass through a record or array type. In Tiger specification, recursions such as

\[
\text{type } b = c \\
\text{type } c = b
\]

are not allowed, while in our implementation it’s allowed since currently we don’t have records and multi-dimensional arrays, which means we don’t have the type recursions for Tiger.

2.1.2 SPIM

We can enter a program and exit correctly. push fp, ra, s0 to stack, and pop them and go back to operating system.

An expression which is an expression sequence surrounded by parenthesis will be translated in order and the overall value is the value of the last expression.

Tiger has 3 kinds of operators, arithmatic, logic and comparison, all can operate on integers and comparisons can work on strings. All work perfectly. We aligned our strings to double word but since We implement comparing strings by loading 1 character each time, the characters in the middle of the strings will still fall unaligned. spim simulator will call kernel code to handle this exception, (eventually the error will be ignored,) which is rather inefficient but we’ll just leave it the way it is for now.

String can be used directly or can be assigned to string variables. Both work. And strings and functions have there real names in assembly code instead of just L1 etc.

Tiger has several predefined functions. we can do type checking for all of them. Since we don’t have the tiger library, I just implemented 2 predefined functions: ‘print’ prints strings and ‘pint’ prints integers. And my implementation of these 2 functions are just to inline them, with no pushing parameters and saving fp, ra because it won’t actually branch to the function body.

We can define functions in tiger too. actual parameters pushed onto stack, jump to function, save fp, ra, point fp to current frame, copy actual parameters and push on the stack as formal parameters, run the program body, save the return value in $v0, pop ra then pop fp basing on current fp instead of sp, and pop the actual parameters out of the stack.

The control flows in-then-else, if-then, while loop and for loop all work. For loop needs some explanation. for an expression like for i:=1 to 10 do exp ‘i’ is an implicitly declared new variable, and the upper and lower bounds are evaluated only once, prior to entering the body of the loop. So 2 local variables are defined for a forloop, i and count. count will be calculated before the iteration and will be decremented 1 each iteration. Both local variables will be popped out the stack when the for loop is done. and tiger has break statement in loops and it works too.

Integer variable declarations push the initialized value of the integer onto the stack, and set the Offset property of the VarIdDec node to its offset compared to current fp, and you get your this offset value by INCLUDING Proc.offset, Program offset, which basically keeps track of the latest biggest offset. every time you push/pop something onto/from the stack, the Offset property of the whole block gets updated too.

User defined types work too, which shouldn’t be a news because good type analysis garentees that.
2.2 Things That Don’t Work Yet

Since tiger’s functions can access variables outside of its own range, i.e. formal parameters and function body, we use fp and attribute 'level' to access those variables. But we didn’t fully implemented the ‘level’ thing yet so currently inside function bodies we can only access formal parameters. Things that we don’t do type analysis don’t work, like records etc. and we didn’t have time to implement arrays.
3 Syntactic Analysis

In this stage of compilation, the text of Tiger source code is read in and transformed into a stream of tokens. In this stage, comments are stripped out, escape sequences are translated, and some basic types are recognized. Tiger’s syntactic analysis process is identical to that of C, with two exceptions. First, Tiger allows nested C-style comments /* like /* this */ */. Second, Tiger allows for linebreaks in string constants, if there is are forward-slash characters immediately before and after the linebreak.

To implement these changes, we simply copied the syntactic analysis C code for C, and modified it.

tiger.gla[2] ≡

```c
id: $[a-zA-Z][a-zA-Z0-9_]* [mkidn]
$/*/ (auxNestedCComment)
Integer:$[0-9]+ [mkidn]
String: $\" (auxTigerString) [tiger_mkstr]
```

This macro is attached to a product file.

This code was copied from Ccomment.c in the Eli packages, and modified to allow for Tiger’s nested C-style comments.

NestedCcomment.c[3] ≡

```c
#include "err.h"
#include "gla.h"
#include "source.h"
#include "tabsize.h"
#include "ScanProc.h"

/**
 ** Returns pointer to first character beyond comment.
 */

char *auxNestedCComment(char *start, int length)
/* Scan a C comment after the opening delimiter,
 * but unlike C, allows for nested comments.
 * On entry
 * start points to the opening delimiter (/*)
 * len=length of the opening delimiter
 * On exit-
 * Ctext points to the character position following the
 * closing delimiter that leaves zero open comments
 ***/
{ register char c;
  register char *p = start + length; /* first char not yet processed */

  for (; ;)
  {
    if (*p == '\0') {
      int current = p - start;
      TokenStart = start = auxNUL(start, current);
      p = start + current;
      if (*p == '\0') {
        message(ERROR,"file ends inside this comment",0,&curpos);
        return p;
      }
    }
  }
```

7
This macro is attached to a product file.

This code was copied from cChStr.c in the Eli packages, and modified to allow for a special escape sequence \__, where __ stands for one or more formatting characters, including whitespace and linebreaks. This sequence is then ignored.

It may seem redundant that the readescape() and auxCChar() are completely unmodified from the original C version of this file. Indeed, only a few lines in the entire file are modified. This is but that is unavoidable. Since there’s no header file with prototypes for the original, there is no way to reference those functions.

TigerChStr.c[4] ≡

```c
#include <stdlib.h>
#include "err.h"
#include "source.h"
#include "gla.h"
#include "csm.h"
#include "tabsize.h"
#include "ScanProc.h"

#define BKSLASH '\\'
#define SQ "\""

static int CharValue; /* global to hold value of escape sequence */

static char *
#include defined(__cplusplus) || defined(__STDC__)
readescape(char *start, int m)
#else
readescape(start, m) char *start; int m;
#endif
/* Advance past an escape, obtaining the value of that escape
 * 0n entry-
 * start addresses the character following the backslash
 * m=0 if no errors are to be reported
 * 1 if errors are to be reported
 * 0n exit-
 * readescape addresses the character following the escape
```


```c
{ register char *p = start;
  register int c = *p++;
  register int count;

  switch (CharValue = c) /* assume simple escape like " or ' */
  {
    /* HEX */
    case 'x':
      CharValue = 0;
      count = 0;
      while (1)
      {
        c = *p++;
        if (!((c >= 'a' && c <= 'f') && ((c >= 'A' && c <= 'F')
            && ((c >= '0' && c <= '9'))))
          { p--; break; }  
        CharValue *= 16;
      
        if (c >= 'a' && c <= 'f')
          CharValue += c - 'a' + 10;
        if (c >= 'A' && c <= 'F')
          CharValue += c - 'A' + 10;
        if (c >= '0' && c <= '9')
          CharValue += c - '0';
        count++;
      }
      if (count == 0 && m) {
        POSITION here = curpos;
        ColOf(here) = start - StartLine;
        message (ERROR, "\x used with no following hex digits",0,&here);
      }
      return p;
    /* OCTAL */
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7':
      CharValue = 0;
      count = 0;
      while (((c <= '7') && (c >= '0') && (count++ < 3))
      {
        CharValue = (((CharValue * 8) + (c - '0')));  
        c = *p++;
      }
      p--; return p;
    /* MISC */
    case '\r':
      if (*p == '\n') p++;
    case '\n': case BKSLASH: case SQ: case '"': case '?':
  }
```
return p;

/* CONTROL */
case 'a': CharValue = '\'07'; return p;
case 'n': CharValue = '\'n'; return p;
case 't': CharValue = '\'t'; return p;
case 'r': CharValue = '\'r'; return p;
case 'f': CharValue = '\'f'; return p;
case 'b': CharValue = '\'b'; return p;
case 'v': CharValue = '\'v'; return p;
}
} /* end switch */

if (c == '\'t') StartLine -= TABSIZE(p - StartLine);

if (m) {
  POSITION here = curpos;
  ColOf(here) = start - StartLine;
  message(NOTE,"unknown escape sequence",0, &here);
}
return p;

/
* auxTigerString
*
* Scan a string literal after the opening double quote.
* *
* On entry, start points to the opening quote, len should be 1.
* On exit, we return the position after the closing quote.
*/

char *
#if defined(__cplusplus) || defined(__STDC__)
auxTigerString(char *start, int len)
#else
auxTigerString( start, len ) char *start; int len;
#endif
{
  register char *p = start + len;
  char delim = *start;
  int escape = 0;

  for ( ; ; ) {
    register char c = *p++;
    if (c == delim) return p;

    if (c == '\'n' || c == '\'r') {
      POSITION here;
      LineOf(here) = LineNum; ColOf(here) = p - StartLine - 1;
      message(ERROR, "String terminated by newline", 0, &here);
      message(ERROR, "Illegal newline in string literal", 0, &curpos);
      return p - 1;
    }
if (c == BKSLASH) {
    if (*p != 'n' && *p != 'r' && *p != 't' && *p != 't')
        p = readescape(p, 1);
    else {
        while (*p != BKSLASH) {
            if (*p == 'n' || *p == 'r') {
                if (*p == '0') {
                    int pSave = p - start;
                    size_t sSave = p - StartLine;
                    TokenStart = start = auxNUL(start, pSave);
                    p = start + pSave; StartLine = start + sSave;
                    if (*p == '0') {
                        message(ERROR, "String terminated by end-of-file", 0, &curpos);
                        return p - 1;
                    }
                }
                LineNum++;
                StartLine = p - 1;
            } else if (*p == 't')
                StartLine -= TABSIZE(p - StartLine);
            else if (*p == ' ') { /* Nothing to do */ }
            else
                message(ERROR, "Non-whitespace character in whitespace escape!",
                        0, &curpos);
            p++;
        }
    p++; // Skip over closing slash
    }
}

if (c == '\t') StartLine -= TABSIZE(p - StartLine);

/*
 * tiger_mkstr
 *
 * Make an internal string value from a character string by
 * collapsing escape sequences.
 *
 * On entry, c points to a character string of length
 * t points to a location containing the initial terminal code
 * On exit, t points to a location containing the correct terminal code
 * an internal string value representing the character string has been
 * stored at the location pointed to by v.
 */

void
#if defined(__cplusplus) || defined(__STDC__)
tiger_mkstr(char *c, int length, int *t, int *v)
#else
tiger_mkstr(c, length, t, v)
#endif
char *c; int length, *t; int *v;
#endif
{
    char *tmp;
    int newlen;

    c++; /* skip leading quote */
    newlen = 0; /* keep track of how long the new string is */
    length -= 2; /* throw away both quotes */

    for (; length > 0; ) {
        if (*c == BKSLASH) {
                while (c[1] != BKSLASH) { length--; c++; }
                length -= 2; /* throw away escaped NLs */
                c += 2;
            } else {
                tmp = readescape(c + 1, 0);
                if (CharValue == 0) {
                    message(ERROR,"Denotation truncated due to embedded \\0",0,&curpos);
                    break;
                }
                obstack_1grow(Csm_obstk, CharValue);
                ++newlen;
                length -= tmp - c;
                c = tmp;
            }
        } else {
            obstack_1grow(Csm_obstk, *c++);
            ++newlen;
            --length;
        }
    }
    obstack_1grow(Csm_obstk, '\0');
    CsmStrPtr = (char *)obstack_finish(Csm_obstk);
    *v = stostr(CsmStrPtr, newlen);
}

/* auxCChar
 * Scan a character literal after the opening single quote.
 * On entry, start points to the opening quote, len should be 1.
 * On exit, we return the position after the closing quote.
 */
char *
#if defined(__cplusplus) || defined(__STDC__)
auxCChar(char *start, int len)
#else
auxCChar(start, len)
#endif
char *start;
int len;
#endif
{
  register char c;
  register char *p = start + len;
  POSITION loc;

  if ((c = *p++) == BKSLASH)
    if (*p == '\n' || *p == '\r') {
      message(ERROR, "newline in character constant", 0, &curpos);
      return p;
    } else if(c == '\n' || c == '\r') {
      message(ERROR, "newline in character constant", 0, &curpos);
      return p - 1;
    } else if(c == SQ) {
      message(ERROR, "Character constant may not be empty", 0, &curpos);
      return p;
    }
  if (*p != '\') {
    loc = curpos; ColOf(loc) += (p - start);
    message(ERROR, "Closing quote required", 0, &loc);
    return p;
  }
  return p + 1;
}

/*
 * tiger_mkchar
 *
 * On entry, c points to a character string of length l
 *  t points to the initial terminal code (not used)
 * On exit,
 *   The location v contains the value of the character literal.
 */

void
#if defined(__cplusplus) || defined(__STDC__)
tiger_mkchar(char *c, int l, int *t, int *v)
#else
tiger_mkchar(c, l, t, v)
#endif
char *c;
int l, *t;
int *v;
#endif

13
char * tmp;
    /* skip leading single quote (') */

c++; if(*c == BKSLASH){
    tmp = readescape(c + 1, 0);
    if( *tmp != SQ)
        message(ERROR,"malformed character constant", 0,&curpos);
    *v = CharValue;
} else
    *v = (int) *c;
}

This macro is attached to a product file.
4 Source Tree Construction

In this stage, the token stream produced by the syntactic analysis stage is transformed into a source tree that better represents the structure of the Tiger programming language.

In our development of source tree specification, we tried to stay as close as possible to what was described by the Tiger specification, without deviating from the logical structure described or introducing additional levels of complexity.

However, in parts of the program, we were forced to deviate from this goal in order to later be able to perform correct name analysis in the next stage of compilation. See especially the definition of the let statement below.

4.1 Abstract Grammar

The abstract grammar is broken up into four different groups; the program as a whole, expressions, declarations, and identifiers. The entire grammar in defined in a single file, shown below.

```
tiger.lido[5] ≡
  Source Tree Program[6]
  Source Tree Expressions[7]
  Source Tree Declarations[16]
  Source Tree Identifiers[20]
```

This macro is attached to a product file.

4.1.1 Program

Valid Tiger programs consist of exactly one expression. Of course, in most programs, this expression will be a let statement or expression sequence, because a program consisting of just one of any other expression generally isn’t very useful.

```
RULE Root: Program ::= exp END;
```

This macro is invoked in definition 5.

4.1.2 Expressions

Tiger defines eight different kinds of expressions; sequences, literals, procedure calls, arithmetic, variables, assignment, flow control, and let.

```
Source Tree Expressions[7] ≡
  Expression Sequence[8]
  Expression Literals[9]
  Expression Procedure Calls[10]
  Expression Arithmetic[11]
  Expression Variables[12]
  Expression Assignment[13]
  Expression Flow Control[14]
  Expression Let[15]
```

This macro is invoked in definition 5.
4.1.2.1 Sequence

An expseq is a semicolon-delimited list of zero or more expressions. Its only valid context is surrounded by parantheses, in which case the entire sequence is treated as a single expression, with its value equal to that of the last expression in the list. An expseq with zero expressions yields no value.

Expression Sequence[8] ≡

RULE Seq: exp ::= '(' expseq ')' END;
RULE Seq0: expseq ::= END;
RULE Seq1: expseq ::= exp END;
RULE seqs: expseq ::= expseq ';' exp END;
RULE ExpSqs: expsqs ::= expsqs END;
RULE ExpSq: expsqs ::= exp END;

This macro is invoked in definition 7.

4.1.2.2 Literals

Literals are allowed only in Tiger’s two predefined types: String and Integer. See the Syntactic Analysis section for details on their parsing.

Expression Literals[9] ≡

RULE integer: exp ::= Integer END;
RULE string: exp ::= String END;

This macro is invoked in definition 7.

4.1.2.3 Procedure Calls

We defined two different, although largely identical forms of procedure call: procedure use and procedure libarray use. This simplifies the task of name analysis for the predefined procedures in the next stage of compilation.

Expression Procedure Calls[10] ≡

RULE FunCall: exp ::= ProcIdUse '(' Actuals ')' END;
RULE FunLibCal: exp ::= ProcLib '(' Actuals ')' END;

This macro is invoked in definition 7.

4.1.2.4 Arithmetic

Arithmetic in Tiger is in common notation, with operators:

'&' bitwise AND, '|' bitwise OR, '+' addition, '-' subtraction, '*' multiplication, '/' division, '=' equality, '<>' non-equality, '>' greater than, '<' less than, '>=' greater than or equal, '<=' less than or equal

Arithmetic expressions take expressions as parameters, so they can be naturally chained together, as well as used seamlessly within the language on different kinds of expressions.

See the concrete grammar for precedence determination.

Expression Arithmetic[11] ≡

RULE calc: exp ::= exp op exp END;
RULE negation: exp ::= op exp END;

This macro is invoked in definition 7.
4.1.2.5 Variables

Variable expressions simply evaluate into the value stored by that variable, or when used as an l-value, evaluate to the storage for that variable.

*Expression Variables*\[12\] \(\equiv\)

\[
\text{RULE ID: } \text{exp ::= VarIdUse END;}
\]

This macro is invoked in definition 7.

4.1.2.6 Assignment

Only l-values can be assigned to, and the only nodes that may be parsed as l-values are variable identifiers or array cells.

*Expression Assignment*\[13\] \(\equiv\)

\[
\text{RULE Assgnmnt: } \text{exp ::= exp ':=' exp END;}
\]

This macro is invoked in definition 7.

4.1.2.7 Flow Control

Flow control in Tiger is similar to C; defining *if-then*, *if-then-else*, *while*, and *for* constructs, the latter two optionally with *break* statements enclosed.

For loops are a little more complicated than the other constructs, because it must have two different scopes that get merged; its variable declaration part and its 'do' expression.

*Expression Flow Control*\[14\] \(\equiv\)

\[
\begin{align*}
\text{RULE IfThnEls: } & \quad \text{exp ::= 'if' exp 'then' exp 'else' exp END;} \\
\text{RULE IfThen: } & \quad \text{exp ::= 'if' exp 'then' exp END;} \\
\text{RULE Iteration: } & \quad \text{exp ::= iteration END;} \\
\text{RULE WhileLoop: } & \quad \text{iteration ::= 'while' exp 'do' exp END;} \\
\text{RULE forLoop: } & \quad \text{iteration ::= expRS END;} \\
\text{RULE ForLoop: } & \quad \text{expRS ::= 'for' idDecRE ':=' exp 'to' exp 'do' expRE END;} \\
\text{RULE ExpForRE: } & \quad \text{expRE ::= exp END;} \\
\text{RULE Break: } & \quad \text{exp ::= 'break' END;}
\end{align*}
\]

This macro is invoked in definition 7.

4.1.2.8 Let

In use, the *let* construct allows the programmer to declare a set of variables, functions and types which are then available for use inside the context of its 'in' block. It also allows for recursive and mutually recursive definitions, so long as they are in an uninterrupted sequence.

In implementation, *Let* is considerably more complex than any other element in the language. The Tiger language requires narrowing scopes with each variable declaration and with each *sequence* of either type or function declarations. So, we wrote the grammar so that a let block with multiple such variables or sequences is equivalent to multiple nested let blocks, each with exactly one variable declaration or sequence of type or function declarations.
In addition to that, we have to do a lot of management of variable scoping, to merge the ’let’ and ’in’ blocks scopes, and allowing nested scopes access to variables, but not parent scopes.

Expression Let[15] ≡

```
RULE LetRSc: exp ::= 'let' letRSc 'end' END;
RULE LetRS: exp ::= 'let' letRS 'end' END;
RULE LetExpSeq: exp ::= 'let' 'in' expseq 'end' END;
RULE: letRSc ::= typDecSeq letRSc END;
RULE: letRS ::= funDecSeq letRSc END;
RULE: letRSc ::= typDecSeq letRS END;
RULE: letRSc ::= funDecSeq letRS END;
RULE: letRS ::= typDecSeq 'in' expseq END;
RULE: letRS ::= funDecSeq 'in' expseq END;
RULE: letRS ::= vardec letRE END;
RULE: letRE ::= letRS END;
RULE: letRE ::= typDecSeq letRSc END;
RULE: letRE ::= typDecSeq letRS END;
RULE: letREE: letRE ::= 'in' expseq END;
```

This macro is invoked in definition 7.

4.1.3 Declarations

Tiger allows for three kinds of declarations; variables, types and functions. These declarations may, if the user desires, overwrite the predefined types and functions.

Source Tree Declarations[16] ≡

```
  Declarations Variables[17]
  Declarations Types[18]
  Declarations Functions[19]
```

This macro is invoked in definition 5.

4.1.3.1 Variables

The Tiger specification allows for both explicitly typed variable declarations, as well as variable declarations in which the type can be inferred from initializing assignment. However, due to the complexities of implementing implicitly typed declarations, we decided to remove that and require that all variable declarations be explicitly typed.

Declarations Variables[17] ≡

```
RULE VarDecTypd: vardec ::= 'var' idDecRE ':' typIdUse ':=' exp END;
```

This macro is invoked in definition 16.
4.1.3.2 Types

Three kinds of type declarations are possible in Tiger; aliases, arrays and records.

*Declarations Types*\[18\] \(\equiv\)

\[
\text{RULE TypDec: typDecSeq ::= tydec END;}
\]
\[
\text{RULE TypDecRec: typDecSeq ::= tydec typDecSeq END;}
\]
\[
\text{RULE TyDec: tydec ::= 'type' typIdDec '=' typDec END;}
\]
\[
\text{RULE TypIsId: typDec ::= typIdUse END;}
\]
\[
\text{RULE TypIsArry: typDec ::= ArrayDec END;}
\]

This macro is invoked in definition 16.

4.1.3.3 Functions

Functions are declared in a form like `function myFunc(i:int, text:string):int`, which declares a function named `myFunc`, taking two parameters, an integer named `i` and a string named `text`, and returning an integer.

The return type is optional. A function declaration with a return type must return a value of that type, but a declaration with no type must not return anything.

*Declarations Functions*\[19\] \(\equiv\)

\[
\text{RULE FunDec: funDecSeq ::= fundec END;}
\]
\[
\text{RULE FunDecRec: funDecSeq ::= fundec funDecSeq END;}
\]
\[
\text{RULE ProcDec: fundec ::= 'function' ProcIdDec Proc END;}
\]
\[
\text{RULE proc: Proc ::= '(' formals ')' '=' exp END;}
\]
\[
\text{RULE procTypd: Proc ::= '(' formals ')' ':' typIdUse '=' exp END;}
\]
\[
\text{RULE Actls: Actuals LISTOF Actual END;}
\]
\[
\text{RULE Actl: Actual ::= exp END;}
\]
\[
\text{RULE Formals: formals LISTOF formal END;}
\]
\[
\text{RULE Formal: formal ::= idDec ':' typIdUse END;}
\]

This macro is invoked in definition 16.

4.1.4 Identifiers

Identifiers in Tiger follow the same rules as C; an alpha character followed by any number of alpha, numeric or underscore characters. There are three classes of identifiers; variables, types and procedures.

Variables can be declared in for loops, let blocks, and in lists of formal parameters in function declarations. Types and procedures, on the other hand, can only be declared in let blocks.

*Source Tree Identifiers*\[20\] \(\equiv\)

\[
\text{RULE IdDec: idDec ::= id END;}
\]
RULE IdDecRE: idDecRE ::= idDec COMPUTE
    idDecRE.Key = idDec.Key;
END;
RULE VrIdUse: VarIdUse ::= id END;
RULE PrcIdDec: ProcIdDec ::= id END;
RULE PrcIdUse: ProcIdUse ::= id END;
RULE TypIdDec: typIdDec ::= id END;
RULE TypIdUse: typIdUse ::= id END;

This macro is invoked in definition 5.

4.2 Concrete Grammar

We wrote the concrete grammar strictly to provide operator precedence and associativity, and to solve conflicts in our abstract grammar.

\[ \text{tiger.con}^{[2]} \equiv \]

Sequencing: '(/[ expseq ']')' .
expseq: exp '$)' '$end' .

FuncCall: id '(/[ Actuals ']')' / ProcLib '(/[ Actuals')' .
Actuals: [exp // ','] .

exp: Assignment / IfThenElse / IfThen / WhileLoop / ForLoop / Break / Let / ArrayCreation / Calculation .
Calculation: Calculation '& Comparison / Calculation '|' Comparison / Calculation .
Comparison: Arithmetic Comparitor Arithmetic / Arithmetic .
Arithmetic: Arithmetic '+' Term / Arithmetic '-' Term / Term .
Term: Term '*' Operand / Term '/' Operand / Operand .
Operand: Integer / String / Sequencing / FuncCall .
Operand: '-' Operand / lvalue .
lvalue: id '$[' .
lvalue: id '[ Subs ']' .
Comparitor: '=' / '<>' / '>' / '<' / '>= / '<= .

ArrayCreation: id '[ Subs ']' 'of' exp .

Assignment: lvalue ':=' exp .
IfThenElse: 'if' exp 'then' exp 'else' exp .
IfThen: 'if' exp 'then' exp '$else'.
WhileLoop: 'while' exp 'do' exp .
ForLoop: 'for' id '://' exp 'to' exp 'do' exp .
Break: 'break' .
Let: 'let' letRSc 'end' / 'let' letRS 'end' / 'let' 'in' expseq 'end' .

letRE: letRScF .
letRScF: funDecSeq letRSc .
letRScF: funDecSeq letRS .
letRScF: funDecSeq 'in' expseq .
4.3 Concrete to abstract mapping

The map simply maps all of the rules in the concrete grammar into the abstract grammar.

\[
\text{tiger.map}[22] \equiv
\]

MAPSYM

\[
\begin{align*}
\text{exp} & ::= \text{Sequencing} \mid \text{FuncCall} \mid \text{Calculation} \mid \text{Comparison} \mid \text{Arithmetic} \mid \text{Term} \\
\text{Operand} & \mid \text{lvalue} \mid \text{Assignment} \mid \text{IfThenElse} \mid \text{IfThen} \mid \text{Break} \mid \text{Let} \\
\text{iteration} & ::= \text{WhileLoop} \\
\text{expRS} & ::= \text{ForLoop} \\
\text{op} & ::= \text{Comparitor} \\
\text{letRSc} & ::= \text{letRScF}
\end{align*}
\]

This macro is attached to a product file.
5 Name Analysis

In this stage of compilation, the completed source tree is analyzed to determine the relationships between identifiers, taking into account scope and other language rules. We also check for use of undeclared names and multiply defined names.

5.1 Modules

Tiger has two name spaces; one for functions and variables, and the other for types. So, we instantiate two AlgScope and AlgRangeSeq modules, one for each name space.

Name Analysis Modules[23] ≡

$/Name/AlgScope.gnrc +instance=Var :inst
$/Name/AlgRangeSeq.gnrc +instance=Var :inst
$/Name/AlgScope.gnrc +instance=Typ :inst
$/Name/AlgRangeSeq.gnrc +instance=Typ :inst

This macro is invoked in definition 1.

5.2 Specification

name.lido[24] ≡

Name Analysis Utility Symbols[25]
Name Analysis Functions[26]
Name Analysis Misc[27]
Name Analysis Let[28]
Name Analysis Types[29]

This macro is attached to a product file.

5.2.1 Name Checking

These symbols There were a number of symbols we defined in the name analysis specification that do not quite fit into any category, or were defined for testing purposes only. idPrint checks for undefined names on usage, while idDecPrint checks for multiply defined names on declaration.

Name Analysis Utility Symbols[25] ≡

CHAIN Defined: VOID;

RULE Root: Program ::= exp COMPUTE
    CHAINSTART exp.Defined=1;
END;

SYMBOL idPrint COMPUTE
    IF(EQ(THIS.Key,NoKey),
        message(NOTE,"Not defined\n",0,COORDREF));
END;

SYMBOL idDecPrint: DoneSet: VOID;
SYMBOL idDecPrint COMPUTE
    SYNT.DoneSet = SetDef(SYNT.Key, 1, 2) <- INH.Defined;
    SYNT.Defined = IF(EQ(GetDef(THIS.Key,0),2),
        message(NOTE,"Multiply defined\n",0,COORDREF)) <- SYNT.DoneSet;
END;

SYMBOL idSym COMPUTE
    SYNT.Sym = TERM;
END;

SYMBOL idDec INHERITS VarIdDefScope, idSym, idDecPrint END;
SYMBOL VarIdUse INHERITS VarIdUseEnv, idSym, idPrint END;

This macro is invoked in definition 24.

5.2.2 Functions

Proc starts a new range, i.e. ProcIdDec belongs to outer scope while formals and function body belong to the same range scope.

If return type is not specified, the function is a procedure and the function body must produce no value.

Predefined functions are defined as operators, with signature specified, and the operator indication is the function name.

Name Analysis Functions\[26\] ≡

    SYMBOL Proc INHERITS VarRangeScope END;
    SYMBOL ProcIdDec INHERITS VarIdDefScope, idSym, idDecPrint END;
    SYMBOL ProcIdUse INHERITS VarIdUseEnv, idSym, idPrint END;

This macro is invoked in definition 24.

5.2.3 For

These symbols are used to manage the scoping rules of 'for' loops. A 'for' loop has its variable declaration and its 'do' block in joined scopes.

Name Analysis Misc\[27\] ≡

    SYMBOL expRS INHERITS VarRangeSequence END;
    SYMBOL idDecRE INHERITS VarRangeElement END;
    SYMBOL expRE INHERITS VarRangeElement END;

This macro is invoked in definition 24.

5.2.4 Let

These symbols are used to manage the scoping rules of 'let' constructs. A 'let' construct also has it's declaration block and its 'do' block in joined scopes.

Name Analysis Let\[28\] ≡

    SYMBOL letRSc INHERITS VarRangeScope, TypRangeScope END;
    SYMBOL letRS INHERITS VarRangeSequence, TypRangeSequence END;
    SYMBOL letRE INHERITS VarRangeElement, TypRangeElement END;

This macro is invoked in definition 24.
5.2.5 Types

Name Analysis Types[29] ≡

\[
\begin{align*}
\text{SYMBOL} & \text{typIdDec INHERITS TypIdDefScope, idSym, idDecPrint END;} \\
\text{SYMBOL} & \text{typIdUse INHERITS TypIdUseEnv, idSym, idPrint END;}
\end{align*}
\]

This macro is invoked in definition 24.
6 Type Analysis

In this stage of compilation, now that we have the completed source tree with all of its names checked and their references resolved, we need to analyze types to determine how the program will handle data and to make sure that types are consistent throughout.

6.1 Modules

Type Analysis Modules[30] ≡

$/Type/Typing.gnrc :inst
$/Type/Expression.fw
$/Type/PreDefOp.gnrc +referto=(op.d) :inst

This macro is invoked in definition 1.

6.2 Inherited Symbols

Type Analysis Inherited Symbols[31] ≡

SYMBOL exp INHERITS ExpressionSymbol END;
SYMBOL op INHERITS OperatorSymbol END;

SYMBOL idDec INHERITS TypedDefId END;
SYMBOL idDecRE INHERITS TypedDefinition END;
SYMBOL VarIdUse INHERITS TypedUseId, ChkTypedUseId END;
SYMBOL typIdDec INHERITS TypeDefDefId, ChkTypeDefDefId END;
SYMBOL typIdUse INHERITS TypeDefUseId, ChkTypeDefUseId END;

This macro is invoked in definition 32.

6.3 Specification

Type analysis is broken up into ten parts for each element of the language with types; inherited symbols, literals, operators, functions, arrays, variables, expressions sequences, type declarations, flow control, and the let statement.

type.lido[32] ≡

Type Analysis Inherited Symbols[31]
Type Analysis Integer and String Literals[33]
Type Analysis Operators[34]
Type Analysis Functions[35]
Type Analysis Arrays[36]
Type Analysis Variables[37]
Type Analysis Expression Sequence[38]
Type Analysis Type Declaration[39]
Type Analysis Flow Control[40]

This macro is attached to a product file.
6.3.1 Literals

Tiger allows only string and integer literals. These are simply a primary context with their respective types.

Type Analysis Integer and String Literals[33] ≡

RULE integer: exp ::= Integer COMPUTE
    PrimaryContext ( exp, intType );
END;

RULE string: exp ::= String COMPUTE
    PrimaryContext ( exp, strType );
END;

This macro is invoked in definition 32.

6.3.2 Operators

Operators are quite simple, as well. Negation is a monadic context of its one parameter, and all other operators are a dyadic context with both of its operators.

Type Analysis Operators[34] ≡

RULE calc: exp ::= exp op exp COMPUTE
    DyadicContext ( exp[1], op, exp[2], exp[3] );
END;

RULE negation: exp ::= op exp COMPUTE
    MonadicContext ( exp[1], op, exp[2] );
END;

This macro is invoked in definition 32.

6.3.3 Functions

Functions are more complicated. We have to take into account return types as well as formal vs actual parameters.

Return types are handled in rules proc and procTypd: a procedure with no return type is required to evaluate a void-type value, where a procedure with a type is required to evaluate to the specified type. Parameter types are checking occurs in rules FunCall and FunLibCal, and is largely handled for us by the type checking module.

Type Analysis Functions[35] ≡

.symbol Actuals INHERITS ArgumentListRoot END;
symbol Actual INHERITS ArgumentListElem END;
symbol Proc INHERITS ProcedureDenotation END;
symbol formals INHERITS ParameterListRoot END;
symbol formal INHERITS ParameterListElem END;
symbol fundec : ind: DefTableKey;
symbol ProcLib INHERITS OperatorSymbol END;

26
This macro is invoked in definition 32.

6.3.4 Arrays

Arrays also have more complicated type analysis.

The ArryDec and BndDumb rules are responsible for associating a type with an array declaration.

The ArryCrt rule simply denotes the array as a primary context with the specified type.

The ArryCrt rule handles array creation, denoting a call context, making sure the subscript is the correct type (integer), the 'of' expression evaluates to the correct type, and that the specified array actually exists.

The ArryFld rule, on the other hand, handles array use. It also denotes a call context and makes sure that the array exists.

Type Analysis Arrays[36] \[36\]
6.3.5 Variables

Variables represent a primary context with the type that was specified on their declaration. Assignment is also a primary context, but it returns a void type.

Type Analysis Variables[37] ≡

RULE ID: exp ::= VarIdUse
PrimaryContext ( exp, VarIdUse.Type );
6.3.6 Expression Sequence

The structure of the expression sequence rules is slightly complicated, because the language specified that
the value of the entire sequence is equal to the value of the last expression in the sequence. There is no very
simple way to reference the last item in a list, so we set it up so the last item passes its own type up to the
scope enclosing itself and the previous expression, until the type of the entire list’s scope is equal to the type
of the last item.

Type Analysis Expression Sequence\[38\] ≡

\[
\text{SYMBOL } \text{expseq INHERITS ExpressionSymbol END;}
\]
\[
\text{SYMBOL } \text{expseq INHERITS ExpressionSymbol END;}
\]

RULE Seq: exp ::= ‘( expseq )’ COMPUTE
TransferContext ( exp, expseq );
END;

RULE Seq0: expseq ::= COMPUTE
PrimaryContext ( expseq, voidType );
END;

RULE Seq1: expseq ::= exp COMPUTE
TransferContext ( expseq, exp );
END;

RULE seqs: expseq ::= expseq ;' exp COMPUTE
TransferContext ( expseq[1], exp );
END;

RULE ExpSqs: expseq ::= expseq ',; exp COMPUTE
TransferContext ( expseq[1], exp );
END;

RULE ExpSq: expseq ::= exp END;

This macro is invoked in definition 32.
6.3.7 Type Declaration

A type declaration simply takes the type from its right hand side and assigns it to its left hand side.

*Type Analysis Type Declaration* [39] ≡

```plaintext
RULE TyDec: tydec ::= 'type' typIdDec '=' typDec COMPUTE
    typIdDec.Type = typDec.Type;
END;

SYMBOL typDec : Type : DefTableKey;

RULE TypIsId: typDec ::= typIdUse COMPUTE
    typDec.Type = typIdUse.Type;
END;
```

This macro is invoked in definition 32.

6.3.8 Flow Control

**If-then-else** blocks require that the type of its 'then' expression and 'else' expression are of equal type, and that the entire block should evaluate to that type. In this case, we always transfer type from the context of the 'then' expression, but since we require that the type of 'then' and 'else' expressions are equal, this should always produce a correct value. The condition expression is required to be of integer type.

Similarly, **while** loops require integer type for its condition expression, and void type for the 'do' expression, so the entire loop evaluates to void type as specified by the language.

Again similarly, **for** loops require void type for its expression, but must evaluate its 'for' block to a variable declaration, and both bounds to integers.

**break** is a primary context of void type, useful only for breaking out of loops.

*Type Analysis Flow Control* [40] ≡

```plaintext
RULE IfThnEls: exp ::= 'if' exp 'then' exp 'else' exp COMPUTE
    TransferContext(exp[1], exp[3]);
    exp[2].Required = intType;
    exp[4].Required = exp[3].Type;
END;

RULE IfThen: exp ::= 'if' exp 'then' exp COMPUTE
    PrimaryContext(exp[1], voidType);
    exp[2].Required = intType;
    exp[3].Required = voidType;
END;

SYMBOL iteration INHERITS ExpressionSymbol END;
SYMBOL expRS INHERITS ExpressionSymbol END;

RULE Iteration: exp ::= iteration COMPUTE
    TransferContext(exp, iteration);
```

30
END;

RULE forLoop: iteration ::= expRS  COMPUTE
    TransferContext(iteration, expRS);
END;

RULE WhileLoop: iteration ::= 'while' exp 'do' exp
COMPUTE
    PrimaryContext(iteration, voidType);
    exp[1].Required = intType;
    exp[2].Required = voidType;
END;

SYMBOL expRE INHERITS ExpressionSymbol END;

RULE ForLoop: expRS ::= 'for' idDecRE ':=' exp 'to' exp 'do' expRE
COMPUTE
    TransferContext(expRS, expRE);
    idDecRE.Type = intType;
    exp[1].Required = intType;
    exp[2].Required = intType;
    expRE.Required = voidType;
END;

RULE ExpForRE: expRE ::= exp
COMPUTE
    PrimaryContext(expRE, voidType);
END;

RULE Break: exp ::= 'break'
COMPUTE
    PrimaryContext(exp, voidType);
END;

This macro is invoked in definition 32.
7 Predefined Language Elements

The tiger specification lists two predefined types, several predefined operators for both of those types, and several predefined functions.

\texttt{tiger.pdl}[41] \equiv

\begin{verbatim}
Predefined Keys Types[45]
Predefined Keys Operators[48]
Predefined Keys SPIM[52]
\end{verbatim}

This macro is attached to a product file.

\texttt{op.d}[42] \equiv

\begin{verbatim}
Predefined Operator Indications[46]
Predefined Operator Signatures[47]
Predefined Function Indications[50]
Predefined Function Signatures[51]
\end{verbatim}

This macro is attached to a product file.

7.1 Modules

We load two separate instances of the predefined type modules; one for type analysis that handles just the predefined types, and one for name analysis that handles operators and functions.

\texttt{Predefinition Modules}[43] \equiv

\begin{verbatim}
$/Name/PreDefine.gnrc +instance=Typ +referto=id :inst
$/Name/PreDefId.gnrc +instance=Typ +referto=(type.d) :inst
$/Name/PreDefine.gnrc +instance=Var +referto=id :inst
$/Name/PreDefId.gnrc +instance=Var +referto=(name.d) :inst
\end{verbatim}

This macro is invoked in definition 1.

7.2 Specification

7.2.1 Types

7.2.1.1 Identifiers

Here we define the identifiers that come predefined in the language; 'int' for integer type and 'string' for string types. Tiger has just two predefined types; integers and strings.

\texttt{type.d}[44] \equiv

\begin{verbatim}
PreDefKey ( "int", intName)
PreDefKey ( "string", strName)
\end{verbatim}

This macro is attached to a product file.
7.2.1.2 Keys

Here we declare keys that will be used to represent the predefined types internally, specify a few of their properties, and associate them with their default indications. We define both integer and string pointer length to be 32 bits, equal to the word length of the MIPS architecture.

**Predefined Keys Types**[45] ≡

```c
intType -> IsType = { 1 };
intType -> Size = { 4 };  /* suppose int number is 4 bytes */
strType -> IsType = { 1 };
strType -> Size = { 4 };  /* suppose address is 4 bytes */
intName -> Defer = {intType};
strName -> Defer = {strType};
voidType-> IsType = { 1 };
```

ArryCmpnt: DefTableKey;
Def: int;
SpimOp: DefTableKey;

This macro is invoked in definition 41.

7.2.2 Operators

Here we declare the indications for all of our predefined operators, both arithmetic and logical (which in Tiger are equivalent, since boolean values are expressed with integers).

7.2.2.1 Indications

**Predefined Operator Indications**[46] ≡

```c
PreDefInd ('&', op, andInd)
PreDefInd ('|', op, orInd)
PreDefInd ('+', op, plusInd)
PreDefInd ('-', op, minusInd)
PreDefInd ('*', op, mulInd)
PreDefInd ('/', op, divInd)
PreDefInd ('=', op, equInd)
PreDefInd ('<>', op, neqInd)
PreDefInd ('>', op, grtInd)
PreDefInd ('<', op, lesInd)
PreDefInd ('>=', op, geqInd)
PreDefInd ('<=', op, leqInd)
```

This macro is invoked in definition 42.
7.2.2.2 Signatures

Here we define the proper usage for all of the predefined operators with a list of valid signatures, and associate them with the indications we just specified and the keys we’re about the specify.

Predefined Operator Signatures[47] ≡

- PreDefOpr (andInd, iAndOp, (intType,intType): intType )
- PreDefOpr (orInd, iOrOp, (intType,intType): intType )
- PreDefOpr (plusInd, iAddOp, (intType,intType): intType )
- PreDefOpr (minusInd, iSubOp, (intType,intType): intType )
- PreDefOpr (minusInd, iNegOp, (intType ): intType )
- PreDefOpr (mulInd, iMulOp, (intType,intType): intType )
- PreDefOpr (divInd, iDivOp, (intType,intType): intType )
- PreDefOpr (equInd, iEqOp, (intType,intType): intType )
- PreDefOpr (neqInd, iNeqOp, (intType,intType): intType )
- PreDefOpr (grtInd, iGrtOp, (intType,intType): intType )
- PreDefOpr (lesInd, iLesOp, (intType,intType): intType )
- PreDefOpr (geqInd, iGeqOp, (intType,intType): intType )
- PreDefOpr (leqInd, iLeqOp, (intType,intType): intType )
- PreDefOpr (equInd, sEqOp, (strType,strType): intType )
- PreDefOpr (neqInd, sNeqOp, (strType,strType): intType )
- PreDefOpr (grtInd, sGrtOp, (strType,strType): intType )
- PreDefOpr (lesInd, sLesOp, (strType,strType): intType )
- PreDefOpr (geqInd, sGeqOp, (strType,strType): intType )
- PreDefOpr (leqInd, sLeqOp, (strType,strType): intType )

This macro is invoked in definition 42.

7.2.2.3 Keys

Here we define all of the keys representing the operators, and associate them with SPIM operations.

Predefined Keys Operators[48] ≡

- iAndOp -> SpimOp = { andOp };
- iOrOp -> SpimOp = { orOp };
- iAddOp -> SpimOp = { addOp };
- iSubOp -> SpimOp = { subOp };
- iNegOp -> SpimOp = { negOp };
- iMulOp -> SpimOp = { mulOp };
- iDivOp -> SpimOp = { divOp };
- iEqOp -> SpimOp = { seqOp };
- iNeqOp -> SpimOp = { sneOp };
- iGrtOp -> SpimOp = { sgtOp };
- iLesOp -> SpimOp = { sltOp };
- iGeqOp -> SpimOp = { sgeOp };
- iLeqOp -> SpimOp = { sleOp };
- sEqOp -> SpimOp = { eqOp };
- sNeqOp -> SpimOp = { neqOp };
- sGrtOp -> SpimOp = { grtOp };
- sLesOp -> SpimOp = { lesOp };
- sGeqOp -> SpimOp = { geqOp };
- sLeqOp -> SpimOp = { leqOp };

This macro is invoked in definition 41.
7.2.3 Functions

7.2.3.1 Indications

These symbols are used to map the predefined functions so that code can be generated from them.

\[
\text{name.d} \equiv
\]

\[
\begin{array}{l}
\text{PreDefSym ("print", printName)} \\
\text{PreDefSym ("pint", pintName)} \\
\text{PreDefSym ("main", mainName)} \\
\end{array}
\]

This macro is attached to a product file.

The function indications are defined just for the sake of typechecking. They take no part in producing code.

\[
\text{Predefined Function Indications} \equiv
\]

\[
\begin{array}{l}
\text{PreDefInd ('print', ProcLib, printInd)} \\
\text{PreDefInd ('flush', ProcLib, flushInd)} \\
\text{PreDefInd ('getchar', ProcLib, getcharInd)} \\
\text{PreDefInd ('ord', ProcLib, ordInd)} \\
\text{PreDefInd ('chr', ProcLib, chrInd)} \\
\text{PreDefInd ('size', ProcLib, sizeInd)} \\
\text{PreDefInd ('substring', ProcLib, substringInd)} \\
\text{PreDefInd ('concat', ProcLib, concatInd)} \\
\text{PreDefInd ('not', ProcLib, notInd)} \\
\text{PreDefInd ('exit', ProcLib, exitInd)} \\
\text{PreDefInd ('pint', ProcLib, pintInd)} \\
\end{array}
\]

This macro is invoked in definition 42.

7.2.3.2 Signatures

Here we define the proper usage for all of the predefined functions with a list of valid signatures, and associate them with the indications we just specified and the keys we’re about the specify.

\[
\text{Predefined Function Signatures} \equiv
\]

\[
\begin{array}{l}
\text{PreDefOpr (printInd, printOper, (strType): voidType)} \\
\text{PreDefOpr (flushInd, flushOper, ( ): voidType)} \\
\text{PreDefOpr (getcharInd, getCharOper, ( ): strType)} \\
\text{PreDefOpr (ordInd, ordOper, (strType): intType)} \\
\text{PreDefOpr (chrInd, chrOper, (intType): strType)} \\
\text{PreDefOpr (sizeInd, sizeOper, (strType): intType)} \\
\text{PreDefOpr (substringInd, substringOper, (strType, intType, intType): strType)} \\
\text{PreDefOpr (concatInd, concatOper, (strType, strType): strType)} \\
\text{PreDefOpr (notInd, notOper, (intType): intType)} \\
\text{PreDefOpr (exitInd, exitOper, (intType): voidType)} \\
\text{PreDefOpr (pintInd, pintOper, (intType): voidType)} \\
\end{array}
\]

This macro is invoked in definition 42.
8 spim

Predefined Keys SPIM[52] ≡

```c
syscallOp -> Pname = {"syscall"};
movOp -> Pname = {"move"};

Size: int; /* of bytes */
Offset: int [Add];
Level: int;
StrLbl: DefTableKey;

void Add ( DefTableKey key, TYPE val )
{
    if (!ACCESS)
        VALUE = 0;
    else
        VALUE = VALUE + val;
}
```

This macro is invoked in definition 41.

spim.lido[53] ≡

```c
ATTR level: int;
ATTR offset: DefTableKey;

RULE: RootSpim ::= Program $Target COMPUTE
    Target.GENTREE = Spim(define_routine(NewKey(),Program.spim));
    PTGOutFile("tiger.s",Target.Code);
END;

ATTR spim: NODEPTR;
CHAIN theChain: VOID;

RULE PItems: Item ::= myItems END;

RULE My13Item: myItems ::= Item Item Item Item Item Item Item
    Item Item Item Item Item Item END;

RULE My9Item: myItems ::= Item Item Item Item Item
    Item Item Item Item Item Item END;

RULE fpPushPop: Item ::= Mop Memory COMPUTE
    Item.Instr = PTGcpu1m (GetPname ( Mop, "BAD" ),30,Memory.Code);
END;
RULE raPushPop: Item ::= Mop Memory COMPUTE
    Item.Instr = PTGcpu1m (36
```
GetPname ( Mop, "BAD" ),
31,
Memory.Code
);
END;
RULE s0PushPop: Item ::= Mop Memory COMPUTE
  Item.Instr = PTGcpu1m ( GetPname ( Mop, "BAD" ),
  16,
  Memory.Code
 );
END;
RULE v0PushPop: Item ::= Mop Memory COMPUTE
  Item.Instr = PTGcpu1m ( GetPname ( Mop, "BAD" ),
  2,
  Memory.Code
 );
END;
RULE movefp: Item ::= Mop Denoter COMPUTE
  Item.Instr = PTGcpu2s ( GetPname ( Mop, "BAD" ), 30, 29, StringTable( Denoter
 ) );
END;
RULE restoresp: Item ::= Mop COMPUTE
  Item.Instr = PTGcpu2s ( GetPname ( Mop, "BAd" ), 29, 30, StringTable ( 0[55] )
 );
END;

RULE Root: Program ::= exp COMPUTE
  CHAINSTART exp.theChain = ResetOffset ( Program.offset, 12 );
  Program.offset = NewKey();
  Program.level = 1;
  Program.spim = MkPItems ( NoPosition,
    MkMy13Item ( NoPosition,
      MkMy1Item ( NoPosition, define_label ( .mainkey ) ),
      MkStack ( NoPosition,
        subOp, IdnNumb ( 0 , 12 )
      ),
      MkfpPushPop ( NoPosition, swOp,
        based_memory ( absolute_address ( IdnNumb ( 0, 12 ) ), 29 )
    ),
    MkraPushPop ( NoPosition, swOp,
      based_memory ( absolute_address ( IdnNumb ( 0, 8 ) ), 29 )
    ),
    Mks0PushPop ( NoPosition, swOp,
      based_memory ( absolute_address ( IdnNumb ( 0, 4 ) ), 29 )
    )
  );
  37
Mkmovelp ( NoPosition,  
    addOp, IdnNumb ( 0, 12 )  
  ),  
sw_iStore ( exp.spim, S[61] ),  
MkVOPushPop ( NoPosition, lwOp, S[61] ),  
Mkrestoresp ( NoPosition, addOp ),  
Mks0PushPop ( NoPosition, lwOp,  
    based_memory ( absolute_address ( IdnNumb ( 0, NEG(8) ) ), 30 ) ),  
MkraPushPop ( NoPosition, lwOp,  
    based_memory ( absolute_address ( IdnNumb ( 0, NEG(4) ) ), 30 ) ),  
MkfpPushPop ( NoPosition, lwOp,  
    based_memory ( absolute_address ( 0 ), 30 ) ),  
MkMyReturn ( NoPosition, jrOp )
)) <- .s;
/*
Program.spim = MkPItems ( NoPosition,  
MkMy9Item ( NoPosition,  
    MkMy1Item ( NoPosition, define_label ( .mainkey ) ),  
    MkStack ( NoPosition,  
        subOp, IdnNumb ( 0, 12 )  
    ),  
    MkfpPushPop ( NoPosition, swOp,  
        based_memory (  
            absolute_address ( IdnNumb ( 0, 12 ) ),  
            29  
        )  
    ),  
    MkraPushPop ( NoPosition, swOp,  
        based_memory (  
            absolute_address ( IdnNumb ( 0, 8 ) ),  
            29  
        )  
    ),  
    Mks0PushPop ( NoPosition, swOp,  
        based_memory (  
            absolute_address ( IdnNumb ( 0, 4 ) ),  
            29  
        )  
    ),  
    Mkmovelp ( NoPosition,  
        addOp, IdnNumb ( 0, 12 )  
    ),  
    sw_iStore ( exp.spim, S[61] ),  
    MkprintCode ( NoPosition, t[60] ),  
    Mksyscall ( NoPosition )
)) <- .s;*/

.mainkey = NewKey();  
.s = ResetSym ( .mainkey, mainName );

END;
ATTR mainkey: DefTableKey;

SYMBOL exp COMPUTE
    SYNT.spim = li_iRegi(1);
END;

RULE Seq: exp ::= '(' expseq ')' COMPUTE
    exp.spim = expseq.spim;
END;

RULE NullIntReg: IntReg ::= END;

RULE Seq0: expseq ::= COMPUTE
    expseq.spim = MkNullIntReg(NoPosition);
END;

RULE Seq1: expseq ::= exp COMPUTE
    expseq.spim = exp.spim;
END;

RULE seqs: expseq ::= expsqs COMPUTE
    expseq.spim = MkiRegii(NoPosition, expsqs.seq, expsqs.spim);
END;

SYMBOL expsqs: seq: NODEPTR;

RULE iRegii: IntReg ::= myItems IntReg COMPUTE
    IntReg[2].reg = IntReg[1].reg;
END;

RULE MyItems: myItems ::= myItems myItem END;
RULE MyItem: myItems ::= Item END;
RULE My1Item: myItems ::= Item END;
RULE MyNoItem: myItems ::= END;
RULE FromItems: myItems ::= Item Item Item Item Item Item Item Item Item
    END;

SYMBOL myItems INHERITS Coded END;
SYMBOL myItem INHERITS Coded END;

RULE ExpSqs: expsqs ::= expsqs ';' exp COMPUTE
    expsqs[1].spim = exp.spim;
    expsqs[1].seq = MkMyItems(NoPosition, expsqs[2].seq, MkMyItem(NoPosition, ignore_int(expseq[2].spim)));
END;

RULE ExpSq: expsqs ::= exp COMPUTE
    expsqs.spim = exp.spim;
    expsqs.seq = MkMyNoItem(NoPosition);
END;

RULE integer: exp ::= Integer COMPUTE
    exp.spim = li_iRegi(Integer);
END;

RULE MyItemDn: myItem ::= Label Denoter COMPUTE
myItem.Instr=PTGStr(
   PTGDirective(".data",PTGNull()),
   PTGDirective(".align",PTGNumber(StringTable(2[2[57]]))),
   PTGLabelDef(StringTable(GenSym(Label))),
   PTGDirective(".asciiz",PTGSame(StringTable(Denoter))),
   PTGDirective(".text",PTGNull()));

END;

SYMBOL exp: StartOfStr: DefTableKey;
SYMBOL exp COMPUTE
   SYNT.StartOfStr = NoKey;
END;

RULE string: exp ::= String COMPUTE
   exp.StartOfStr = NewKey();
   .set = IF ( exp.Sym,
      ResetSym ( exp.StartOfStr, exp.Sym ) );
   exp.spim = MkRegii(NoPosition,
      MkMyItems(NoPosition, MkMyNoItem(NoPosition),
      MkMyItemDn(NoPosition, exp.StartOfStr, String ) ),
      MḵstrAsActual ( NoPosition,
      based_memory(syman(based address(exp.StartOfStr),0))
    ) <= .set;
END;

RULE jal: IntReg ::= Denoter COMPUTE
   IntReg.Instr = PTGcpu0s ( GetPname ( jalOp, "BAD" ),
      StringTable ( Denoter ) );
END;

RULE func: IntReg ::= myItems IntReg IntReg Item END;

RULE move: IntReg ::= COMPUTE
   IntReg.Instr = PTGcpu2 ( GetPname ( moveOp, "BAD" ),
      IntReg.reg,
      2 );
END;

RULE FunCall: exp ::= ProcIdUse '( Actuals )' COMPUTE
   exp.spim = Mkfunc ( NoPosition,
      Actuals.spim,
      Mkjal ( NoPosition, ProcIdUse.Sym ),
      Mkmove ( NoPosition ),
      MkStack ( NoPosition, addOp, IdnNumb ( 0, .size ) ) );
   .size = CONSTITUENTS Actual.size WITH ( int, ADD, IDENTICAL, ZERO );
   exp.theChain = AddOffset ( INCLUDING ( Proc.offset, Program.offset ), NEG ( .size ) )
    <= Actuals.theChain;
SYMBOL Actuals: toprint: int;
SYMBOL Actuals COMPUTE
  INH.toprint = 0;
END;

RULE inlineP: IntReg ::= myItems Item IntReg END;
RULE syscall: IntReg ::= COMPUTE
  IntReg.Instr = PTGcpu0s (
    GetPname ( syscallOp, "BAD" ),
    StringTable ( 0 )
  );
END;

RULE printCode: Item ::= Denoter COMPUTE
  Item.Instr = PTGcpu1s (
    GetPname ( liOp, "BAD" ),
    2, StringTable ( Denoter )
  );
END;

SYMBOL ProcLib: Sym :int;
SYMBOL ProcLib COMPUTE
  SYNT.Sym = 0;
END;

RULE: ProcLib ::= 'print' COMPUTE
  ProcLib.Sym = printName;
END;

RULE: ProcLib ::= 'pint' COMPUTE
  ProcLib.Sym = pintName;
END;

RULE FunLibCal: exp ::= ProcLib '(' Actuals ')' COMPUTE
  Actuals.toprint = IF ( EQ ( ProcLib.Sym, printName ), 1, 2 );
  exp.spim =
    MkinlineP ( NoPosition,
      Actuals.spim,
      MkprintCode ( NoPosition,
        IF ( EQ ( Actuals.toprint, 1 ), 4[58], 1[56] ) ),
      Mksyscall ( NoPosition ) );
END;

ATTR tLbl: DefTableKey;
ATTR neqLbl: DefTableKey;
ATTR ltLbl: DefTableKey;
ATTR zero, one, two: NODEPTR;
ATTR loads1: NODEPTR;
ATTR loads2: NODEPTR;
ATTR loadi: NODEPTR;
ATTR s1Mem, rMem, s2Mem: NODEPTR;
ATTR iMem: NODEPTR;

RULE calc: exp ::= exp op exp COMPUTE
exp[1].spim = IF ( EQ ( exp[2].Type, intType ),
MkiRegrr ( NoPosition,
  GetSpimOp ( op.Oper, NoKey ),
  exp[2].spim,
  exp[3].spim
 ),
MkiRegii ( NoPosition,
MkFromItems ( NoPosition,
    ignore_int ( exp[2].spim ),
    ignore_int ( exp[3].spim ),
    sw_iStore ( .zero, .iMem ),
define_label ( .loopLbl ),
do_branch ( bne_Branchrr ( .loads1, .loads2 ), .neqLbl ),
do_branch ( beqz_Branchr ( .loads1 ), .tLbl ),
  sw_iStore ( add_iRegrr ( .loadi, .one ), .iMem ),
j_Jump ( .loopLbl ),
define_label ( .tLbl ),
  sw_iStore ( .one, rMem ),
j_Jump ( .endLbl ),
define_label ( .neqLbl ),
do_branch ( blt_Branchrr ( .loads1, .loads2 ), .ltLbl ),
  sw_iStore ( .two, .rMem ),
j_Jump ( .endLbl ),
define_label ( .ltLbl ),
  sw_iStore ( .zero, .rMem ),
define_label ( .endLbl )
  ),
MkiRegrr ( NoPosition,
GetSpimOp ( op.Oper, NoKey ),
lw_iLoad ( .rMem ),
  .one
 )
).
loads1 = lw_iLoad ( .s1Mem );
loads2 = lw_iLoad ( .s2Mem );
loadi = lw_iLoad ( .iMem );
s1Mem = indexed_memory ( symbolic_address ( exp[2].StartOfStr ), .loadi );
s2Mem = indexed_memory ( symbolic_address ( exp[3].StartOfStr ), .loadi );
rMem = $S[61]; /* result variable */
iMem = $S[61];
loopLbl= NewKey();
endLbl = NewKey();
tLbl = NewKey(); /* terminal label */
egqLbl = NewKey(); /* not equal label */
ltLbl = NewKey(); /* less than label */
zero = li_iRegi ( 0[55] );
one = li_iRegi ( 1[56] );
two = li_iRegi ( 2[57] );
RULE negation: exp ::= op exp COMPUTE
    exp[1].spim = MkRegr (NoPosition,
        GetSpimOp (op.Oper, NoKey),
        exp[2].spim);
END;

RULE strAsActual: IntReg ::= Memory COMPUTE
    IntReg.Instr = PGcpu1m (GetPname (laOp, "BAD"),
        4, Memory.Code);
END;

RULE ID: exp ::= VarIdUse COMPUTE
    exp.spim = IF (exp.lv,.
        spim,
        IF (EQ (INCLUDING (Actuals.toprint, ROOTCLASS.toprint), 1),
            MkstrAsActual (NoPosition, .spim),
            lw_iLoad(.spim)));
    .spim = IF (EQ (VarIdUse.Type, strType),
        based_memory (symbolic_address (GetStrLbl (VarIdUse.Key, 0)),
            0),
        /* else is intType */
        based_memory (absolute_address (IdnNumb (0, NEG (GetOffset (VarIdUse.Key, 0))))),
            30)) <- exp.theChain;
    exp.StartOfStr = IF (EQ (VarIdUse.Type, strType),
        GetStrLbl (VarIdUse.Key, 0), NoKey);
    exp.theChain = exp.StartOfStr;
END;

SYMBOL exp: lv: int;
SYMBOL exp COMPUTE
    INH.lv = 0;
END;

RULE Assgnmnt: exp ::= exp ':==' exp COMPUTE
    exp[2].lv = 1;
    exp[1].spim = sw_iStore(exp[3].spim, exp[2].spim);
END;

ATTR elseLbl: DefTableKey;
ATTR endLbl: DefTableKey;
RULE IRIfThnEl: IntReg ::= Item IntReg Item Item IntReg Item  
COMPUTE
  IntReg[2].reg = IntReg[1].reg;
  IntReg[3].reg = IntReg[1].reg;
END;

RULE IRIfThn: IntReg ::= Item IntReg Item END;

RULE IfThnEl: exp ::= 'if' exp 'then' exp 'else' exp  
COMPUTE
  exp[1].spim = 
    MkIRIfThnEl( NoPosition, 
    do_branch(beqz_Branchr(exp[2].spim), .elseLbl), 
    exp[3].spim, 
    j_Jump(.endLbl), 
    define_label(.elseLbl), 
    exp[4].spim, 
    define_label(.endLbl) );
  .elseLbl = NewKey();
  .endLbl = NewKey();
END;

RULE IfThen: exp ::= 'if' exp 'then' exp  
COMPUTE
  exp[1].spim = 
    MkIRIfThn( NoPosition, 
    do_branch( beqz_Branchr( exp[2].spim ), .endLbl ), 
    exp[3].spim, 
    define_label(.endLbl) );
  .endLbl = NewKey();
END;

RULE IRWhileLoop: IntReg ::= Item Item IntReg Item Item 
COMPUTE
  IntReg[2].reg = IntReg[1].reg;
END;

ATTR loopLbl: DefTableKey;
SYMBOL iteration: endLbl: DefTableKey;
SYMBOL expRS : endLbl: DefTableKey;

RULE Iteration: exp ::= iteration  
COMPUTE
  exp.spim = iteration.spim;
END;

RULE WhileLoop: iteration ::= 'while' exp 'do' exp  
COMPUTE
  iteration.spim = MkIRWhileLoop ( NoPosition, 
  define_label(.loopLbl), 
  do_branch( beqz_Branchr(exp[1].spim), iteration.endLbl), 
  exp[2].spim, 
  j_Jump(.loopLbl), 
  define_label(iteration.endLbl) );
  .loopLbl = NewKey();
  iteration.endLbl = NewKey();
END;
RULE forLoop: iteration ::= expRS COMPUTE
  iteration.spim = expRS.spim;
  iteration.endLbl = expRS.endLbl;
END;

RULE IRForLoop: IntReg ::= myItems Item Item IntReg IntReg Item Item Item END;

ATTR spim2: NODEPTR;
ATTR spim3: NODEPTR;
ATTR count: DefTableKey;
ATTR countaddr: NODEPTR;
ATTR size2: int;

RULE ForLoop: expRS ::= 'for' idDecRE ':=' exp 'to' exp 'do' expRE COMPUTE
  expRS.spim = MkIRForLoop ( NoPosition,
    .spim,
    define_label ( .loopLbl ),
    do_branch ( bltz_Branchr ( .spim3 ), expRS.endLbl ),
    expRE.spim,
    sw_iStore ( sub_iRegrr ( .spim3, .spim2 ), .countaddr),
    j_Jump ( .loopLbl ),
    define_label ( expRS.endLbl ),
    MkStack ( NoPosition, addOp, 8[59] )
  );
  .spim = MkMyItems2 ( NoPosition,
    MkStack ( NoPosition, subOp, IdnNumb ( 0, .size2 ) ),
    sw_iStore ( sub_iRegrr ( exp[2].spim,
      sw_iStore ( exp[1].spim,
        based_memory ( absolute_address ( 8[59] ), 29 ))),
        based_memory ( absolute_address ( 4[58] ), 29 ))
  );
  .count = NewKey();
  .countaddr = based_memory ( absolute_address ( IdnNumb ( 0, NEG ( GetOffset ( .count, 0 ) ) ) ), 30 );
  expRE.theChain = ORDER ( ResetOffset ( idDecRE.Key, GetOffset ( INCLUDING ( Proc.offset, Program.offset ), 0 ) ),
    AddOffset ( INCLUDING ( Proc.offset, Program.offset ), .size ),
    ResetOffset ( .count, GetOffset ( INCLUDING ( Proc.offset, Program.offset ), 0 ) ),
    AddOffset ( INCLUDING ( Proc.offset, Program.offset ), .size ) ) <- expRS.theChain ;
  expRS.theChain = AddOffset ( INCLUDING ( Proc.offset, Program.offset ), NEG ( 8 ) ) <- expRE.theChain;
  .size2 = MUL ( .size, 2 );
  .size = GetSize ( intType, 0 );
  .spim2= li_iRegi( IdnNumb ( 0, 1 ) );
.spim3 = lw_iLoad ( .countaddr );
.loopLbl = NewKey();
expRS.endLbl = NewKey();
END;

SYMBOL expRE COMPUTE
SYNT.spim = MkNullIntReg ( NoPosition );
END;

RULE ExpForRE: expRE ::= exp COMPUTE
expRE.spim = exp.spim;
END;

RULE IRBreak: IntReg ::= Item END;

SYMBOL ROOTCLASS: BADLbl: DefTableKey;
SYMBOL ROOTCLASS: toprint:int;
SYMBOL ROOTCLASS COMPUTE
SYNT.BADLbl = NoKey;
SYNT.toprint = 0;
END;

RULE Break: exp ::= 'break' COMPUTE
exp.spim = MkIRBreak ( NoPosition,
    j_Jump ( INCLUDING ( iteration.endLbl, ROOTCLASS.BADLbl) )
);
END;

RULE LetRSc: exp ::= 'let' letRSc 'end' COMPUTE
letRSc[1].spim = letRSc[2].spim;
END;
RULE LetRS: exp ::= 'let' letRS 'end' COMPUTE
letRS.spim = letRS.spim;
END;
RULE LetExpSeq: exp ::= 'let' 'in' expseq 'end' COMPUTE
exp.spim = expseq.spim;
END;
RULE: letRSc::= typDecSeq letRSc COMPUTE
letRSc[1].spim = letRSc[2].spim;
END;
RULE: letRSc::= funDecSeq letRSc COMPUTE
letRSc[1].spim = MkiRegii ( NoPosition, funDecSeq.spim, letRSc[2].spim );
END;
RULE: letRSc::= typDecSeq letRS COMPUTE
letRSc.spim = letRS.spim;
END;
RULE: letRSc::= funDecSeq letRS COMPUTE
letRSc.spim = MkiRegii ( NoPosition, funDecSeq.spim, letRS.spim );
END;
RULE: letRSc::= typDecSeq 'in' expseq COMPUTE
letRSc.spim = expseq.spim;
END;
RULE: letRSc::= funDecSeq 'in' expseq COMPUTE

letRSc.spim = MkiRegii ( NoPosition, funDecSeq.spim, expseq.spim );
RULE: letRS ::= vardec letRE COMPUTE
letRS.spim = MkiRegii ( NoPosition, vardec.spim, letRE.spim );
RULE: letRE ::= letRS COMPUTE
letRE.spim = letRS.spim;
RULE: letRE ::= letRSc COMPUTE
letRE.spim = letRSc.spim;
RULE: letRE ::= typDecSeq letRSc COMPUTE
letRE.spim = letRSc.spim;
RULE: letRE ::= typDecSeq letRS COMPUTE
letRE.spim = letRS.spim;
RULE: letRE ::= typDecSeq 'in' expseq COMPUTE
letRE.spim = expseq.spim;
RULE letREE: letRE ::= 'in' expseq COMPUTE
letRE.spim = expseq.spim;
SYMBOL funDecSeq COMPUTE
SYNT.spim = MkMyNoItem ( NoPosition );
END;
RULE Stack: Item ::= Mop Denoter COMPUTE
Item.Instr = PTGcpu2s ( GetPname ( Mop, "BAD" ), 29, 29, StringTable( Denoter ) );
END;
ATTR size: int;
RULE MyItems2: myItems ::= Item Item END;
SYMBOL exp: Sym: int;
SYMBOL exp COMPUTE
INH.Sym = 0;
END;
RULE VarDecTypd: vardec ::= 'var' idDecRE ':=' typIdUse ':=' exp COMPUTE
vardec.spim = IF ( EQ ( TransDefer ( typIdUse.Type ), intType ) ),
MkMyItems2 ( NoPosition, 
ignore_int ( sw_iStore ( exp.spim, S[61] ) ),
MkStack ( NoPosition, subOp, IdnNumb ( 0, .size ) ) ),
IF ( EQ ( TransDefer ( typIdUse.Type ), strType ) ),
MkMyItem ( NoPosition, ignore_int ( exp.spim ) ),
MkMyNoItem ( NoPosition )
);
.size = GetSize ( TransDefer ( typIdUse.Type ), 0 );
s = ResetLevel ( idDecRE.Key, INCLUDING ( Proc.level, Program.level ) )
< vardec.theChain;
exp.Sym = idDecRE.Sym <- .s;
vardec.theChain = IF ( EQ ( TransDefer ( typIdUse.Type ), intType ),
ORDER (
  ResetOffset ( idDecRE.Key,
    GetOffset ( INCLUDING ( Proc.offset, Program.offset ), 0 ) ),
  AddOffset ( INCLUDING ( Proc.offset, Program.offset ), .size )
),
IF ( EQ ( TransDefer ( typIdUse.Type ), strType ),
  ResetStrLbl ( idDecRE.Key, exp.StartOfStr ),
  ResetStrLbl ( idDecRE.Key, NoKey )
) <- exp.Sym;
END;

RULE TypDec: typDecSeq ::= tydec END;
RULE TypDecRec: typDecSeq ::= tydec typDecSeq END;

RULE TyDec: tydec ::= 'type' typIdDec '=' typDec END;

RULE TypIsId: typDec ::= typIdUse END;
RULE TypIsArry: typDec ::= ArrayDec END;

RULE FunDec: funDecSeq ::= fundec COMPUTE
  funDecSeq.spim = fundec.spim;
END;

RULE funMyItems: myItems ::= myItems myItems END;
RULE FunDecRec: funDecSeq ::= fundec funDecSeq COMPUTE
  funDecSeq[1].spim = MkfunMyItems ( NoPosition,
    fundec.spim, funDecSeq[2].spim
  );
END;

ATTR fun: DefTableKey;
ATTR endoffun: DefTableKey;

RULE My8Item: myItems ::= Item Item Item Item Item Item myItems Item END;

RULE ProcDec: fundec ::= 'function' ProcIdDec Proc COMPUTE
  fundec.spim = MkMy8Item ( NoPosition,
    j_Jump ( .endoffun ),
    define_label ( .fun ),
    MkStack ( NoPosition,
      subOp, IdnNumb ( 0, 8 )
    ),
    MkfpPushPop ( NoPosition, swOp,
      based_memory ( absolute_address ( IdnNumb ( 0, 8 ) ),
        29 )
    ),
  );
MkraPushPop ( NoPosition, swOp,
  based_memory (absolute_address ( IdnNumb ( 0, 4 ) ),
  29 ) ),
Mkmovefp ( NoPosition,
  addOp, IdnNumb ( 0, 8 ) ),
Proc.spim,
define_label ( .endoffun )
); .fun = NewKey(); ResetSym (.fun, ProcIdDec.Sym);
.endoffun = NewKey();

RULE My6Items: myItems ::= myItems Item Item Item Item Item END;

RULE proc:  Proc ::= '(' formals ')' '=' exp COMPUTE
  Proc.spim = MkMy6Items ( NoPosition, formals.spim,
    ignore_int ( exp.spim ),
    Mkrestoreesp ( NoPosition, addOp ),
    MkraPushPop ( NoPosition, lwOp,
      based_memory ( absolute_address ( IdnNumb ( 0, NEG(4) ) ), 30 ) ),
    MkfPushPop ( NoPosition, lwOp,
      based_memory ( absolute_address ( 0 ), 30 ) ),
    MkMyReturn ( NoPosition, jrOp )
  );
  Proc.level = ADD ( 1, INCLUDING ( Proc.level, Program.level ) ) <- Proc.theChain;
  Proc.offset = NewKey() <- Proc.level;
  formals.theChain = ResetOffset ( Proc.offset, 8 );
END;

RULE My7Items: myItems ::= myItems Item myItems Item Item Item Item END;

RULE MyReturn: Item ::= Mop COMPUTE
  Item.Instr = PTGcpu1 ( GetPname ( Mop, "BAD" ), 31 );
END;

RULE procTypd: Proc ::= '(' formals ')' ':=' typIdUse '=' exp COMPUTE
  Proc.spim = MkMy7Items ( NoPosition, formals.spim,
    sw_iStore ( exp.spim, S[61] ),
    MkvPushPop ( NoPosition, lwOp, S[61] ),
    Mkrestoreesp ( NoPosition, addOp ),
    MkraPushPop ( NoPosition, lwOp,
      based_memory ( absolute_address ( IdnNumb ( 0, NEG(4) ) ), 30 ) ),
    MkfPushPop ( NoPosition, lwOp,
      based_memory ( absolute_address ( 0 ), 30 ) ),
    MkMyReturn ( NoPosition, jrOp )
49
Proc.level = ADD ( 1, INCLUDING ( Proc.level, Program.level ) )
<- Proc.theChain;
Proc.offset = NewKey() <- Proc.level;
formals.theChain = ResetOffset ( Proc.offset, 8 );
END;

RULE Actls:  Actuals LISTOF Actual COMPUTE
Actuals.spim =
CONSTITUENTS Actual.spim
  WITH ( NODEPTR, join2actual, join1formal, joinnoformals);
END;

RULE movetoa0: myItems ::= IntReg COMPUTE
myItems.Instr = PTGcpu2 ( GetPname ( moveOp, "BAD" ),
  4,
  IntReg.reg
);
END;

RULE Actl:  Actual ::= exp COMPUTE
Actual.spim = IF ( INCLUDING Actuals topprint,
  IF ( EQ ( INCLUDING Actuals topprint, 1 ),
    MkMy1Item ( NoPosition, ignore_int ( exp.spim ) ),
    Mkmovetoa0 ( NoPosition, exp.spim )
  ),
  MkMyItems2 ( NoPosition,
    ignore_int ( sw_iStore ( exp.spim, S[61] ) ),
    MkStack ( NoPosition, subOp, IdnNumb ( 0, Actual.size ) )
  ));
Actual.size = GetSize ( exp.Type, 0 );
Actual.theChain = AddOffset ( INCLUDING ( Proc.offset, Program.offset),
  IF ( INCLUDING Actuals topprint, 0, Actual.size )
  )
<- Actual.theChain;
END;

RULE formal2: myItems ::= myItems myItems END;
RULE formal1: myItems ::= myItems END;
RULE noformal: myItems ::= END;

RULE Formals: formals LISTOF formal COMPUTE
formals.spim =
CONSTITUENTS formal.spim
  WITH ( NODEPTR, join2formal, join1formal, joinnoformals);
END;
RULE Formal:  formal ::= idDec ':' typIdUse COMPUTE
formal.spim =
  MkMyItems2 ( NoPosition,
    ignore_int ( sw_iStore ( lw_iLoad ( based_memory ( base_ptr + 2) )
      )
    ));
END;
absolute_address (  
    IdnNumb ( 0,  
        ADD ( SUB ( GetOffset ( INCLUDING Proc.offset, 0 ), 8 ),  
            .size ) ),  
    30 )  
),  
S[61] (  
    MkStack ( NoPosition, subOp, IdnNumb ( 0, .size ) )  
) <- formal.theChain;  
formal.theChain = AddOffset ( INCLUDING Proc.offset, .size )  
<- .set;  
.size = GetSize ( TransDefer ( typIdUse.Type ), 0 );  
.set = ResetOffset ( idDec.Key, GetOffset ( INCLUDING Proc.offset, 0 ) )  
<- formal.spim;  
END;  
RULE IdDec:  
idDec ::= id  
END;  
SYMBOL idDecRE: Sym: int;  
RULE IdDecRE:  
idDecRE ::= idDec  
COMPUTE  
idDecRE.Sym = idDec.Sym;  
END;  
RULE VrIdUse:  
VarIdUse ::= id  
END;  
RULE PrcIdDec:  
ProcIdDec ::= id  
END;  
RULE PrcIdUse:  
ProcIdUse ::= id  
END;  
RULE TypIdDec:  
typIdDec ::= id  
END;  
RULE TypIdUse:  
typIdUse ::= id  
END;  
RULE ArryDec:  
ArrayDec ::= Bound typIdUse  
COMPUTE  
END;  
RULE BndDumb:  
Bound ::= 'array' 'of'  
COMPUTE  
END;  
RULE ArryCrt:  
exp ::= ArrayCreation  
COMPUTE  
END;  
RULE ArryCrt:  
ArrayCreation ::= typIdUse '[ Subs ]' 'of' exp  
COMPUTE  
END;  
RULE ArryFld:  
exp ::= VarIdUse '[ Subs ]'  
COMPUTE  
END;  
RULE subs:  
Subs LISTOF Sub  
END;  
RULE sub:  
Sub ::= exp  
COMPUTE  
END;  
RULE TypIsArry:  
typDec ::= ArrayDec  
COMPUTE
This macro is attached to a product file.

tiger.ptg[54] ≡

  Same: "" $string ""
  Number: $string
  Str: $ $ $ $
  Test: "asdfasdfsadfasdfasfd\n"

This macro is attached to a product file.

0[55] ≡
  IdnNumb ( 0, 0 )

This macro is invoked in definition 53.

1[56] ≡
  IdnNumb ( 0, 1 )

This macro is invoked in definition 53.

2[57] ≡
  IdnNumb ( 0, 2 )

This macro is invoked in definition 53.

4[58] ≡
  IdnNumb ( 0, 4 )

This macro is invoked in definition 53.

8[59] ≡
  IdnNumb ( 0, 8 )

This macro is invoked in definition 53.

t[60] ≡
  IdnNumb ( 0, 10 )

This macro is invoked in definition 53.

S[61] ≡
  based_memory ( absolute_address ( 0 ), 29 )

This macro is invoked in definition 53.

myfac.h[62] ≡

  #define join2formal( x, y )  Mkformal2 ( NoPosition, x, y )
  #define join2actual( x, y )  Mkformal2 ( NoPosition, y, x )
  #define join1formal( x )    Mkformal1 ( NoPosition, x )
  #define joinnoformals()     Mknoformal ( NoPosition )
This macro is attached to a product file.

myfac.head[63] ≡

#include "myfac.h"

This macro is attached to a product file.