TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
BỘ MÔN KHOA HỌC MÁY TÍNH

Trình biên dịch KPL

Tài liệu mô tả chức năng và thiết kế của các bộ Scanner, Parser, Phân tích ngữ nghĩa và Sinh mã

Sinh viên: **Nguyễn Trường Minh**

Lớp: **KSTN-CNTT-K52**

5/15/2011

1. Bộ Scanner

---------- MÔ TẢ CHỨC NĂNG (FUNCTION DESCRIPTION)----------
Bộ scanner hay còn gọi là bộ phân tích từ vựng, có chức năng phân tích các từ tố (token) trong một file mã nguồn.
Từ tố được hiểu là yếu tố cầu thành nhỏ nhất mà một trình biên dịch có thể hiểu được.
Cụ thể hơn, scanner có các nhiệm vụ sau
1. Bỏ qua các ký tự không có nghĩa đối với trình dịch như dấu trắng, tab, xuống dòng, chú thích.
2. Phát hiện các token bao gồm: từ khóa, định danh, các punctuations
3. Chỉ ra các ký tự không nhận dạng được.
Nguyên lý: ô tô mát hữu hạn cho ngôn ngữ chính quy.
Cụ thể hơn [bắt đầu 0] [chuyển trạng thái ứng với các ký tự nhận dạng được] [quay về 0 khi nhận dạng xong token]
Để dễ lập trình ta sử dụng số nguyên để đánh dấu các trạng thái này.
Chú ý: Ident và Keyword chưa được phân biệt ở bộ scanner.

---------- MÔ TẢ THIẾT KẾ (DESIGN DESCRIPTION) ----------

1. Input: một file chứa ký tự, được chuyển vào xử lý dưới dạng character stream
2. Output:
    a. Thành công: dãy các token đã nhận dạng được.
    b. Thất bại: chỉ ra ký tự không nhận dạng được.
3. Thành phần:
    3.1. Makefile
    3.2. scanner.c : tệp chính
    3.3. reader.h, reader.c : đọc mã nguồn dạng characters
    3.4. charcode.h, charcode.c : phân loại ký tự
    3.5. token.h, token.c : nhận dạng tokens
    3.6. error.h, error.c : thông báo lỗi.
4. Nguyên tác hoạt động:
    4.1. Reader.c đọc các ký tự từ file rồi đưa các ký tự này vào một mảng để chương trình sử dụng
    4.2. Token.c đọc từng ký tự của mảng,
    sử dụng ô tô mát hữu hạn để nhận dạng token, bỏ các ký tự không có nghĩa, thông báo lỗi. Đây là tệp quan trọng nhất trong bộ scanner.
        4.2.1. Ứng với giá trị nhận được ở đầu đọc ta chuyển tới các nhánh của ô tô mát.
        (Nhánh 0) Các ký tự kết thúc.
        (Nhánh 1) Loại các ký tự trắng: skipBlank().
        (Nhánh 2) Loại chú thích: skipComment(). Trong KPL: (* *)
        (Nhánh 3) Số: readNumber().
        (Nhánh 4) Tên và từ khóa: readIdentityKeyword(). Ident và Keyword chưa được phân biệt ở bộ scanner.
        (Nhánh 5) Hằng số: readConstantChar().

4.3. Charcode.c ánh xạ các ký tự từ bảng mã ASCII vào các loại ta quan tâm: space, alphabet, digit, punctuation, [],(),unknown.
4.4. Error.c đưa ra thông báo lỗi:
4.4.1. Đang đọc dở Token hay ô tô mát chưa đến trạng thái kết thúc được mà hết mảng ký tự. "End of Document Expected"
4.4.2. Tên quá dài. Do ta chỉ dùng một mảng (thường là 32 ký tự) để lưu định danh. "Identification too long"
4.4.2. Không tìm thấy đóng ngoặc của hằng ký tự. Ví dụ: 'a' thiếu dấu đóng ngoăc. Có hai khả năng: một là ký tự khác ở vị trí này, hai là hết mảng ký tự (4.4.1)
4.4.3. Khi charcode.c trả về unknown. Token sẽ không được tạo ra. "Invalid Symbol".

---

2. Bộ Parser

---------- MÔ TẢ CHỨC NĂNG (FUNCTION DESCRIPTION)----------
Bộ Parser có chức năng kiểm tra cấu trúc ngữ pháp của chương trình.
Cấu trúc ngữ pháp là thứ tự bố trí các từ tố trong một câu.
Cấu trúc ngữ pháp của ngôn ngữ lập trình được biểu diễn dưới dạng cây cú pháp (syntax tree) .
Bộ Parser có các nhiệm vụ sau:
1. Kích ọoạt bộ scanner
2. Xây dựng cây cú pháp từ các token cung cấp bởi bộ scanner.
Nguyên lý: Phương pháp chung là phân tích từ trên xuống (top down).
Ta bắt đầu với các ngôn ngữ không yêu cầu phân tích quay lui, hsy còn gọi là phân tích tiền định.
Cách làm này chỉ có thể được áp dụng với các văn phạm đặc biệt, các văn phạm này là các văn pham LL(k). Văn phạm LL(k) trái là văn phạm mà luật sản xuất được áp dụng được xác định sau khi có k ký tự đầu của suy dẫn trái.
Ví dụ: ngôn ngữ KPL là văn phạm LL(1).
Tuy nhiên trong KPL có hai vị trí vi phạm LL(1) là câu lệnh if-then-else và indent-function trong factor.

---------- MÔ TẢ THIẾT KẾ (DESIGN DESCRIPTION) ----------

1. Input: mảng các token khi xử lý theo lô / current token khi xử lý online
2. Output:
a. Thành công: cây cú pháp
b. Thất bại: chỉ ra lỗi cú pháp
3. Thành phần:
3.1. Makefile
3.2. Bộ Scanner
3.3. parser.h, parser.c
4. Nguyên tác hoạt động:

4.1. Lấy token tiếp theo cần xử lý. Khi xử lý online : getValidToken(). Ta dừng xử lý khi nhận được Token không hợp lệ. Bộ Scanner cần thông báo lỗi, bộ Parser cần cho thoát chương trình.

4.2. Dịch chương trình: compileProgram. Toàn bộ chương trình được chia thành các khối. Mối khối các token mở đầu và kết thúc như một thủ tục bắt buộc.

Ví dụ: một chương trình cần có cấu trúc như sau:

KEYWORD_PROGRAM IDENTITY SYMBOL_SEMOCOLON block() SYMBOL_PERIOD . Program nhan_hai_so_lon; block .

4.3. Dịch block: compileBlock()

Một block có các phần tùy chọn, tức các phần mà người sử dụng có thể viết hoặc bỏ qua.

Thứ tự nhánh phụ thuộc giá trị token: CONST / TYPE / VAR / PROCEDURE / FUNCTION / BEGIN

4.3.1. Khai báo hằng: compileConstDecls().

Hàm dịch khai bao hằng có 3 nhiệm vụ:

(1) Kiểm tra từng token trong thủ tục bắt buộc. Gọi hàm dịch hằng số. H àm dịch hằng sô này có thể phảm làm việc với dấu của số như +,-

(2) Chuyển sang 4.3.2./3./4./. tùy vào token tiếp theo - dựa trên truy cập ngẫu nhiên.

Mô hình chuyển thác nước hay truy cập tuần tự đôi khi cũng được sử dụng. Tuy nhiên lúc này yêu cầu viết chương trình cũng phải tuần tự.

Ví dụ: KPL là văn phạm LL0 nên chỉ cần xét một token tiếp theo để biết nhánh cần lựa chọn.

(3) Thông báo lỗi trong các trường hợp còn lại.

4.3.2. Khai báo các kiểu do người dùng định nghĩa. Ta chỉ có hai kiểu cơ bản là char và integer.

Để xác định kiểu người sử dụng khai báo cần hàm compileType()

Kiểu này là INTEGER / CHAR / IDENT (một kiểu đã định nghĩa trước) / ARRAY (các phần tử kiểu gì? gọi compilteType())

4.3.3. Khai báo biến. Có yêu cầu tìm kiểu,

4.3.4. Khai báo thủ tục (Có thể có nhiều thủ tục).

Trong thủ tục có dãy các tham số hình thức. Để biên dịc ta cần hạm : compileParams().

4.3.5. Khai báo hàm. Khác với khai báo thủ tục, trong khai báo hàm cần có tham số trả về.

Ví duk : KPL yêu cầu kiểu trả về của hàm là kiểu cơ bản.

Một số ngôn ngữ khác cũng chỉ cho phép trả về kiểu cơ bản. Các biến phức được xác định bởi con trỏ.

4.3.6. Phần chương trình chính.

Ví dụ : trong KPL phần chương trình chính có cấu trúc BEGIN statements END

4.4. Dịch statements.

4.4.1. Lệnh gán: compileAssignSt();

Yêu cầu xác định biến / hàm ở vế trái và giá trị biểu thức ở vế phải.

Nếu biến ở dây là biến mảng ta cần có thêm compileIndexes ngay tiếp sau. Trong đó, chỉ số của phần tử trong mảng là một biểu thức.

Một biểu thức được xác định bằng tổng / hiệu của các sô hạng (term). Một số hạng là tích của các nhân tử (factor)

Một nhân tử có thể là một hằng, biến, hàm, hoặc biểu thức khác.
4.4.2. Lệnh gọi thủ tục: compileCallSt();
Yêu cầu xác định tập các tham số, trong đó mỗi tham số là một biểu thức.
4.4.3. Block : compileGroupSt();
4.4.4. Lệnh if: compileIfSt();
Yêu cầu xác định biểu thức điều kiện.
Một biểu thức điều kiện được tạo thành bởi hai biểu thức ở hai vế của một toán tử điều kiện.
Cậu lệnh if-then-else vi phạm điều kiện LL(1).
Trong tình huống này ta gán ELSE cho IF chưa được gán ELSE mà ở gần nó nhất.
4.4.5. Lệnh while: compileWhileSt();
4.4.6. Lệnh for: compileForSt();

4.5. Error.c đưa ra thông báo lỗi: (Invalid : các lỗi về cú pháp)
4.5.1. ERR_END_OF_COMMENT: Ô tô mát chưa kết thúc.
4.5.2. ERR_IDENT_TOO_LONG: Tên quá dài.
4.5.3. ERR_INVALID_CONSTANT_CHAR: Ô tô mát chưa kết thúc.Không tìm thấy đóng ngoặc của hằng ký tự. Ví dụ: 'a' thiếu dấu đóng ngoặc. Có hai khả năng: một là ký tự khác ở vị trí này, hai là hết mảng ký tự (4.4.1)
4.5.4. ERR_INVALID_SYMBOL: Không nhận ra ký tự. Khi charcode.c trả về unknown. Token sẽ không được tạo ra. "Invalid Symbol".
4.5.5. ERR_INVALID_IDENT, "An identifier expected."
4.5.6. ERR_INVALID_CONSTANT, "A constant expected." :
4.5.7. ERR_INVALID_TYPE, ERR_UNDECLARED_TYPE: có 4 loại kiểu-char, integer, array, ident(user-defined).
Nếu trình dịch không tìm thấy kiểu thì báo lỗi undeclared.
Nếu sai về mặt cú pháp (ví dụ: n : 4234 thay vì n : integer) thì báo về invalid.
4.5.8. ERR_INVALID_BASICTYPE, "A basic type expected."
4.5.9. ERR_INVALID_VARIABLE, "A variable expected."
4.5.10. ERR_INVALID_FUNCTION, "A function identifier expected."
4.5.11. ERR_INVALID_PROCEDURE, "A procedure identifier expected."
4.5.12. ERR_INVALID_PARAMETER, "A parameter expected."
4.5.13. ERR_INVALID_STATEMENT, "Invalid statement." Ví dụ: begin KW_PROGRAM
4.5.14. ERR_INVALID_COMPARATOR, "A comparator expected."
4.5.15. ERR_INVALID_EXPRESSION, "Invalid expression." Ví dụ: (5 7)
4.5.16. ERR_INVALID_TERM, "Invalid term." Ví dụ: 5 + -7 - không phát hiện ra số âm.
4.5.17. ERR_INVALID_FACTOR, "Invalid factor." Ví dụ: 5 * -7
4.5.18. ERR_INVALID_LVALUE, "Invalid lvalue in assignment."
4.5.19. ERR_INVALID_ARGUMENTS, "Wrong arguments."

3. Bộ Phân tích ngữ nghĩa

---------- MÔ TẢ CHỨC NĂNG (FUNCTION DESCRIPTION)----------
Chức năng: tìm ra các lỗi sau giai đoạn phân tích cú pháp. Các lỗi này bao gồm:

1. sự tương thích về kiểu (type checking).
   Có hai phương pháp kiểm tra kiểu: tĩnh (compile time, int, float ...) và động (run time , void object trong C)
2. sự tương thích giữa khai báo và sử dụng của hàm , biến , hằng, mảng...
3. Xác định phạm vi tham chiếu của biến.

Nhiệm vụ:

1. Kích hoạt bộ parser.
2. Xây dựng bảng ký hiệu để phục vụ cho việc sinh mã.

Bảng ký hiệu: chứa thông tin về tên, kiểu, phạm vi và kích cỡ bộ nhớ cần phân phối.
---------- MÔ TẢ THIẾT KẾ (DESIGN DESCRIPTION) ----------

1. Input: cây phân tích cú pháp khi xử lý theo lô / nhúng trực tiếp vào bộ parser khi xử lý online
2. Output:
   a. Thành công: chương trình không có lỗi.
   b. Thất bại: thông báo lỗi tương ứng.
3. Thành phần:
   3.1. Makefile.
   3.2. Bộ Parser.
   3.3. Symtab.h, symtab.c : tạo và quản lý bảng ký hiệu hay đúng hơn là bảng các identifier.
   3.4. Semantics.h, semantics.c:
   3.5. Debug.h, debug.c : hỗ trợ debug bằng các hàm in ra đối tượng, kiểu, hằng số, và phạm vi.

I - Xây dựng bảng ký hiệu

- Symtab.h định nghĩa các thành phần sau
  1. Type: có 3 kiểu - INT, CHAR, ARRAY. Nếu là kiểu mảng cần có kích thước và kiểu của các phần tử.
  2. ConstantValue: chứa thông tin về kiểu và giá trị (int hoặc char) của hằng số.
  3. Object: có tên dài không quá MAX_IDENT_LEN. Có 7 loại object:
     - Loại 1: constantAttribute chứa một giá trị hằng.
     - Loại 2: variableAttribute có kiểu và phạm vi xác định.
     - Loại 3: typeAttribute nhận một trong kiểu INT, CHAR, ARRAY.
     - Loại 4: procedureAttribute được xác định bởi danh sách tham số và phạm vi.
     - Loại 5: functionAttribute được xác định bởi danh sách tham số và phạm vi. Thêm vào đó cần biết kiểu trả về.
     - Loại 6: programAttribute được xác định bởi phạm vi. (cần làm rõ hơn)
     - Loại 7: parameterAttribute. Có hai loại tham số là: (loại 1) chứa giá trị và (loại 2) chứa con trỏ. Tham số được xác định dựa vào tên hàm, thứ tự và kiểu.
  4. ObjectNode: danh sách các Object. Cấu trúc này được dùng để lưu danh sách tham số của hàm.

5. Scope: mỗi phạm vi tương ứng với một Object (hàm, thủ tục, program) mà ta gọi owner. Phạm vi bên ngoài của phạm vi này được gọi là outer_scope. Trong scope cần lưu danh sách các Object nằm ở scope đó. Chú ý: không lưu object ở các scope bên trong.
6. SymTab: Bảng ký hiệu của toàn bộ chương trình. Bảng này được đặt tên là tên của chương trình. Trong bảng có lưu danh sách các object ở phạm vi global.
Scope mà trình dịch đang làm việc được symtab theo dõi thông qua thành phần currentScope.

- Symtab.c cài đặt các hàm sau

(1) Các hàm tạo kiểu: int, char, array. Mục đích chính là cấp phát bộ nhớ, gán thành phần xác định kiểu.

Bênh cạnh đó hàm tạo kiểu với tên gọi khác cũng được cài đặt.

Hàm so sánh hai kiểu giúp xác định sự tương thích kiểu.

Đối với kiểu array cần kiểm tra kích thước của mảng và kiểu của từng phần tử.

(2) Các hàm tạo hằng số: int, char.

(3) Các hàm xử lý Object: quản lý phạm vi của biến, hàm và thủ tục dựa trên thành phần scope.

Chú ý: object sau khi được tạo cần được xác định một phạm vi duy nhất hay thuộc một ObjectList duy nhất.

Phạm vi hiện tại cần được thay đổi vào / ra tương ứng với khi vào / ra khỏi một phạm vi cho trước.

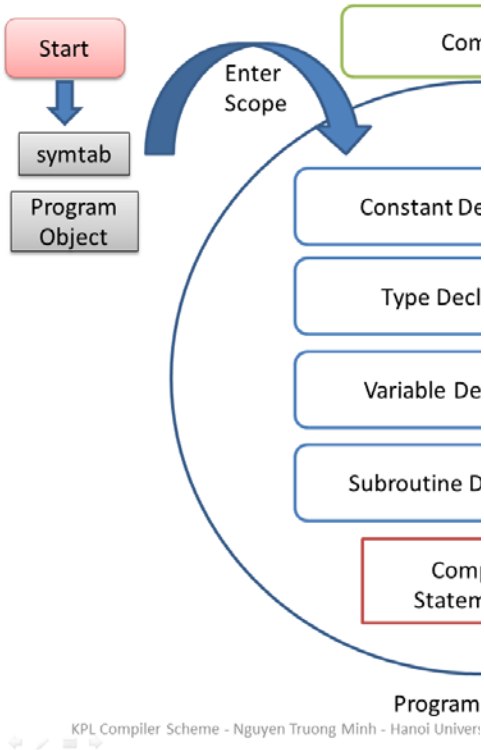
Mỗi khi Object cần tìm không có ở pham vi hiện tại, trình biên dịch tiếp tục tìm kiếm phạm vi ngay bên ngoài.

Công việc tìm kiếm kết thúc khi đến phạm vi toàn cục. program->progAttrs->scope = createScope(program,NULL);

- Xây dựng bảng ký hiêu trong bộ Parser

1. Khởi tạo và giải phóng bảng: initSymTab(); ... ;cleanSymTab();
2. Đối tượng chương trình: khởi tạo tại hàm compileProgram().

Sau khi khởi tạo chương trình, enterBlock chính. Sau khi duyệt xong chương trình exitBlock.



3. Khai báo hằng: các đối tượng hằng số được tạo ra khai báo tại hàm compileBlock().

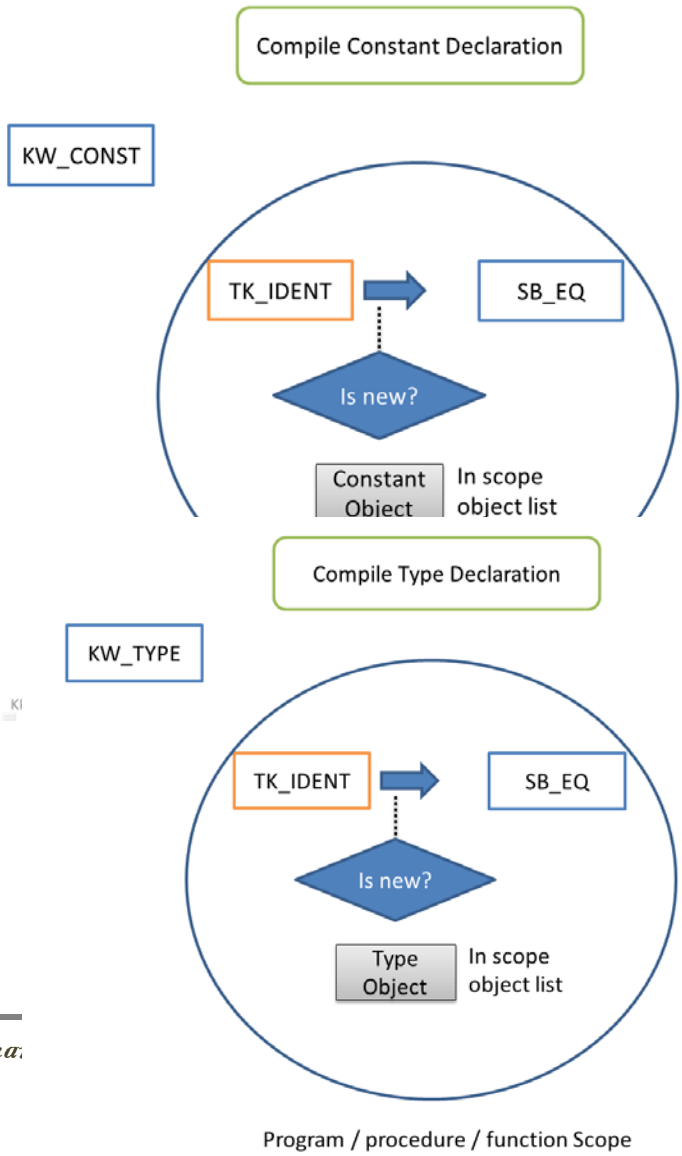
Giá trị hằng số được lấy từ quá trình duyệt giá trị hằng thông qua hàm compileConstant().

Nếu giá trị hằng là một định danh hằng, phải tra bảng ký hiệu để lấy giá trị tương ứng.

Xem case IDENT, hàm compileConstant2(). Nếu không tìm thấy định danh hằng, thông báo lỗi undecleared.

Sau khi duyệt xong hằng số, ta phải đăng ký vào block hiện tại.

4. Khai báo kiểu người dùng định nghĩa: tại hàm compileBlock2().



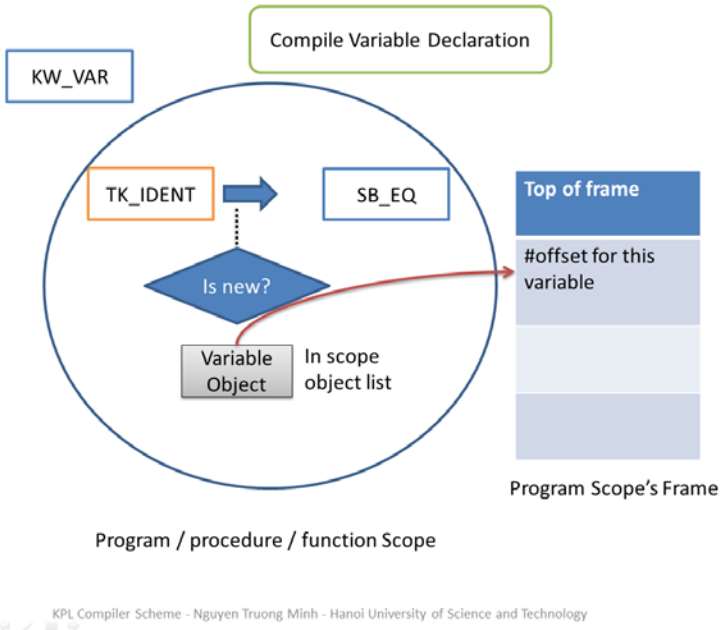
Kiểu thực tế được lấy từ quá trình duyệt kiểu bằng hàm compileType().
Nếu kiểu được định nghĩa thông qua kiểu T , kiểu T cần được khai báo trước. Ngược lại,thông báo lỗi undecleared.
Sau khi duyệt xong kiểu, ta phải đăng ký vào block hiện tại.

5. Khai báo biến: tại hàm compileBlock3().

Kiểu của biến được lấy từ quá trình duyệt kiểu.

Lưu trữ phạm vi hiện tại để phục vụ sinh mã sau này.

Sau khi duyệt xong một biến, ta phải đăng ký vào block hiện tại.



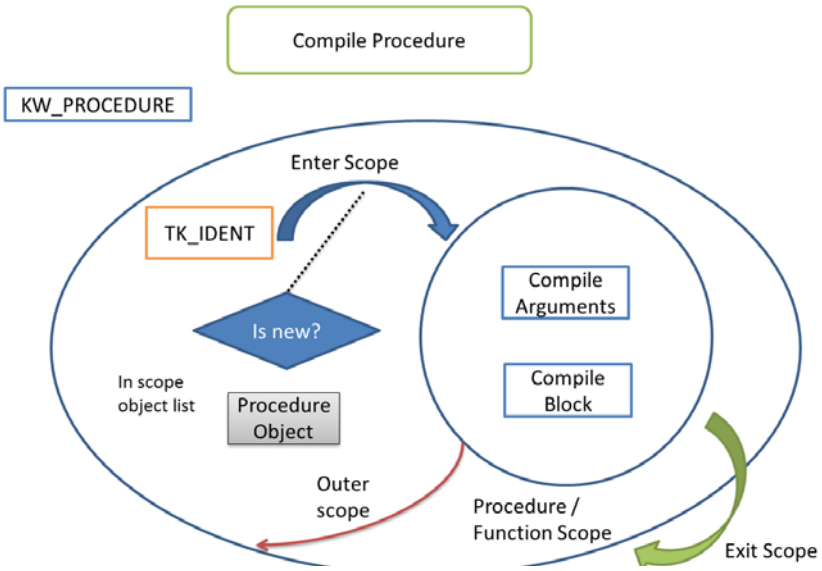
KPL Compiler Scheme - Nguyen Truong Minh - Hanoi University of Science and Technology

6. Khai báo hàm: tại hàm compileFuncDecl().
Cần lưu đối tượng hàm vào block hiện tại, tránh nhầm với phạm vi của hàm.
Các thuộc tính được cập nhật bao gồm:

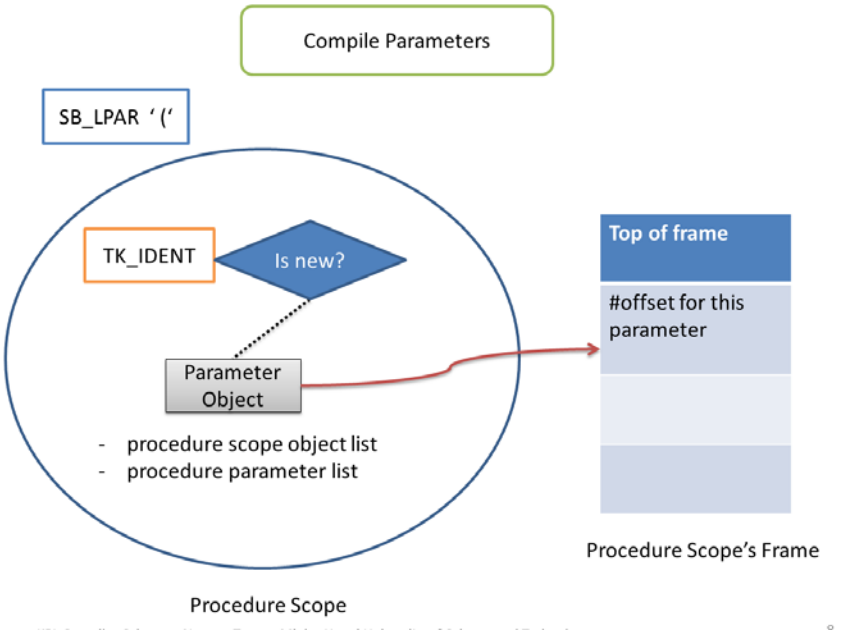
6.1. Danh sách tham số: compileParams(). Có hai loại tham sô: PARAM_VALUE, PARAM_REFERENCE.Mỗi khi duyệt xong một tham số, ta phải đăng ký vào hàm hiện tại.

6.2. Kiểu dữ liệu trả về: compileBasicType().

7. Khai báo thủ tục: tại hàm compileProcDecl().
Cần lưu đối tượng thủ tục vào block hiện tại, tránh nhầm với phạm vi của thuc tục.
Danh sách tham số: compileParams(). Có hai loại tham sô: PARAM_VALUE, PARAM_REFERENCE.
Mỗi khi duyệt xong một tham số, ta phải đăng ký vào thủ tục hiện tại.



8. Khai báo tham số hình thức: compileParam().
Có hai loại tham sô: tham biến - PARAM_VALUE, và tham trị - PARAM_REFERENCE.
Đối tượng tham số được đăng ký vào cả paramList và phạm vi hiện tại (sử dụng trong phạm vi hàm / thủ tục).



II - Kiểm tra sự trùng lặp và phạm vi tham chiếu.
1. Kiểm tra tên hợp lệ
- Một đối tượng có tên hợp lệ nếu nó chưa từng được sử dụng trong phạm vi hiện tại.
Hàm checkFreshIdent() kiểm tra điều kiện trên.
Việc kiểm tra được thực hiện khi xảy ra các sự kiện sau:
1.1. Khai báo hằng: việc kiểm tra một hằng số đã khai báo được thực hiện khi có tham chiếu tới hằng số đó.
Nếu hằng không được định nghĩa ở phạm vi hiện tại thì phải tìm ở phạm vi rộng hơn.
1.2. Kiểm tra kiểu đã khai báo.
Nếu kiểu không được định nghĩa ở phạm vi hiện tại thì phải tìm ở phạm vi rộng hơn.
1.3. Kiểm tra biến đã khai báo: được thực hiện khi có tham chiếu tới biến.

Nếu biến không được định nghĩa ở phạm vi hiện tại thì phải tìm ở phạm vi rộng hơn.
Khi một định danh xuất hiện ở bên trái của biểu thức gán hoặc ở factor, định danh đó có thể tương ứng là
- tên hàm hiện tại
- một biến đã khai báo. Nếu biến khai báo có kiểu mảng, sau tên biến phải có chỉ số của mảng.

Cần phân biệt biến với tham số và tên hàm hiện tại.
1.4. Kiểm tra hàm đã khai báo.
Nếu hàm không được định nghĩa ở phạm vi hiện tại thì phải tìm ở phạm vi rộng hơn.
Một số hàm toàn cục: READC, READI
1.5. Kiểm tra thủ tục đã khai báo.
Nếu thủ tục không được định nghĩa ở phạm vi hiện tại thì phải tìm ở phạm vi rộng hơn.
Một số thủ tục toàn cục: WRITEI, WRITEC, WRITELN.

---

4. Bộ Sinh mã

---------- MÔ TẢ CHỨC NĂNG (FUNCTION DESCRIPTION)----------

Chức năng: biến đổi cây phân tích cú pháp không có lỗi thành chuỗi các lệnh có thể thực thi được trên máy đích.

Tuy nhiên ta chưa sinh mã đích ngay bởi việc sinh mã đích ngay lập tức có nhiều nhược điểm.

- Thứ nhất chương trình chỉ được cài đặt phù hợp với một máy cụ thể.
Nếu chuyển sang máy khác ta lại phải chỉnh sửa mã nguồn. [portability]
- Thứ hai trình gỡ rối phức tạp bởi luôn phải tham chiếu code nguồn và mã máy,
- Thứ ba việc tối ưu hóa phải được thực hiện trên máy tính cụ thể,

và do đó các phương pháp nghiên cứu chung không hăn luôn phát huy hiệu quả

Do đó trình biên dịch sẽ dịch mã nguồn thành một chương trình tương đương viết trên mã trung gian, một loại mã dành cho các máy tính trừu tượng. Các loại mã trung gian phổ biến là: ký pháp Balan hậu tố, mã 3 địa chỉ.

Ở đây ta sử dụng mã 3 địa chỉ. Mỗi lệnh trong chương trình chứa không quá 3 địa chỉ toán hạng. Sau đó từ chương trình dưới dạng mã 3 địa chỉ, trình biên dịch sinh ra mã assembly.

Nhiệm vụ:
1. Xây dựng máy ngăn xếp.
1.1. Máy ngăn xếp là một hệ thống tính toán. Nó bao gồm hai vùng nhớ chính:
- Khối lệnh: chứa mã thực thi của chương trình.
- Ngăn xếp: lưu trữ các kết quả trung gian, biến, phục vụ vào ra hàm/thủ tục.

1.2. Thanh ghi: PC(Program Counter):
- con trỏ trỏ tới lệnh đanh thực thi trên bộ đệm chương trình (khối lệnh)

- B(base): con trỏ trỏ tới địa chỉ gốc của vùng nhớ cục bộ, hay là địa chỉ đầu của ngăn xếp.
- T(top): con trỏ trỏ tới đỉnh ngăn xếp.

1.3. Bản hoạt động (activation record)

- RV (return value): lưu trữ giá trị trả về của hàm. Đối với thủ tục, RV được bỏ qua.
- DL (dynamic link): kết nối máy ngăn xếp hiện tại với máy gọi nó.
Dynamic link giữa caller và callee là quan hệ cha-con.
Ta sử dụng DL để khôi phục ngữ cảnh của caller.
- RA (return address): địa chỉ trả về của hàm / thủ tục.
- SL (static link): kết nối scope hiện tại với scope bên ngoài nó.
Static link giữa scope bên trong và scope outer là quan hệ cha-con.
Ta sử dụng SL để truy cập các biến phi cục bộ.

2. Xây dựng bảng ký hiệu

2.1. Bổ sung thông tin cho biến
2.2. Để đơn giản hóa, mỗi giá trị integer/char đều chiếm từ một từ 4 bytes.
2.3. Thứ tự trên một frame như sau:
0: RV; 1: DL; 2: RA; 3: SL; từ 4 đền 4+k-1: k tham số; từ 4+k đến 4+k+n-1 : n biến cục bộ
2.4. SizeOfType: có ba trường hợp INT, CHAR, ARRAY: ArraySize* ElemenType
2.5. Khai báo Object

2.5.1. Khai báo biến: xác định phạm vi, vị trí trên frame. Tăng kích thước frame theo type của Object
2.5.2. Khai báo tham số: xác định phạm vi, vị trí trên frame. Tăng kích thước frame theo type của Object
Xác định hàm/thủ tục yêu cầu tham số này.
2.6. Kiểm tra một biến, hàm, thủ tục, program xem đã được khai báo chưa
2.7. Kiểm tra tương thích kiểu trong phép gán, điều kiện, truyền tham số.

3. Sinh mã trung gian

3.1. Tất cả được lưu trên codeBlock. Chỉ có một lần tạo codeBlock mà thôi.
Bởi chỉ có initCodeBuffer gọi createCodeBlock. Trong khi đó chỉ có một lần gọi initCodeBuffer ở hàm main().

---------- MÔ TẢ THIẾT KẾ (DESIGN DESCRIPTION) ----------

1. Input: cây phân tích cú pháp không có lỗi.
2. Output:

a. Thành công: chương trình thực thi trên máy hoặc code assembly
b. Thất bại: thông báo lỗi vào ra

3. Thành phần:

3.1. Makefile
3.2. Bộ Phân tích Ngữ nghĩa
3.3. instructions.h, instruction.c: sinh mã cho lệnh.
3.4. codegen.h, codegen.c: bố trí các lệnh trên code buffer.

4. Nguyên tác hoạt động:

4.1. Khởi động chương trình, xử lý tham số vào cho trình dịch

4.2. Khởi tạo bảng ký hiệu.
4.3. Kiểm tra cú pháp rồi tạo program Object. Tiếp theo vào trong program Block, hay chuyển phạm vi hiện tại vào phạm vi của program vừa tạo.
4.4. Error.c đưa ra thông báo lỗi:
4.4.1. Đang đọc dở Token hay ô tô mát chưa đến trạng thái kết thúc được mà hết mảng ký tự. "End of Document Expected"
4.4.2. Tên quá dài. Do ta chỉ dùng một mảng (thường là 32 ký tự) để lưu định danh. "Identification too long"
4.4.2. Không tìm thấy đóng ngoặc của hằng ký tự. Ví dụ: 'a' thiếu dấu đóng ngoặc. Có hai khả năng: một là ký tự khác ở vị trí này, hai là hết mảng ký tự (4.4.1)
4.4.3. Khi charcode.c trả về unknown. Token sẽ không được tạo ra. "Invalid Symbol".

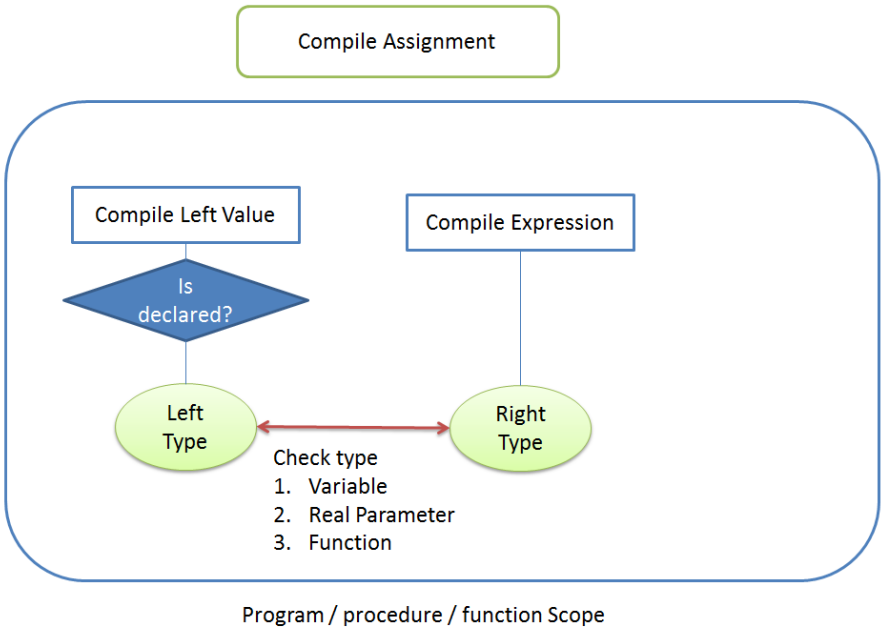
I - Xây dựng máy ngăn xếp

Instruction.h
1. Op_Code: chứa các lệnh của mã ba địa chỉ.
1.1. Các lệnh load: địa chỉ, dữ liệu, hằng số, vùng nhớ xác định qua con trỏ
1.2. Lệnh thêm / xóa ô nhớ trên ngăn xếp: tăng / giảm đỉnh ngăn xếp.
1.3. Lệnh điều khiển con đếm chương trình: jump, jump với điều kiện true/false
1.4. Lệnh dừng chương trình: halt.
1.5. Lệnh cất dữ liệu từ ngăn xếp vào bộ nhớ: store
1.6. Các lệnh phục vụ hàm và thủ tục: gọi hàm/thủ tục và bố trí bản ghi hoạt động, thoát hàm/thủ tục
1.7. Các hàm/thủ tục vào ra: đọc ký tự, số nguyên; viết ký tự, số nguyên; thêm dòng mới
1.8. Các lệnh toán học: cộng, trừ, nhân, chia, đảo dấu.
1.9. Lệnh sao chép dữ liệu: copy
1.10. Các lệnh so sánh: bằng, lớn hơn, nhỏ hơn
1.11. Break point: hỗ trợ debug.
2. Instruction: một lệnh sử dụng mã ba địa chỉ.
3. CodeAddress: địa chỉ code của hàm, thủ tục, hoặc chương trình.
4. CodeBlock: chứa một mảng code có sức chứa maxsize phần tử và hiện đang chứa codeSize phần tử.

Instruction.c
1. CreateCodeBlock: tạo code block với sức chứa theo yêu cầu.
2. EmitCode: thêm lệnh vào code block.
3. Các lệnh sinh code dựa trên Op_Code.
Chú ý đối với các lệnh yêu cầu 1 tham số, DC_VALUE được thêm vào vị trí của p. Đối với các lệnh không yêu cầu tham số, DC_VALUE được thêm vào vị trí của p và q.

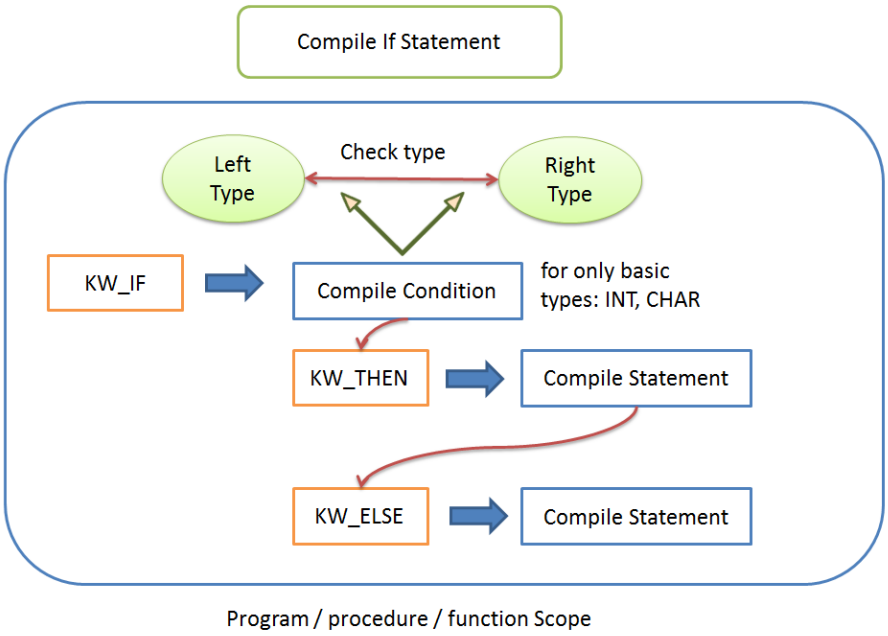
II - Sự tương thích kiểu
Cần kiểm tra sự tương thích kiểu trong

1. Lệnh gán: vế trái là một biến, hàm. Vế phải là một biểu thức, hàm, hằng số, biến.



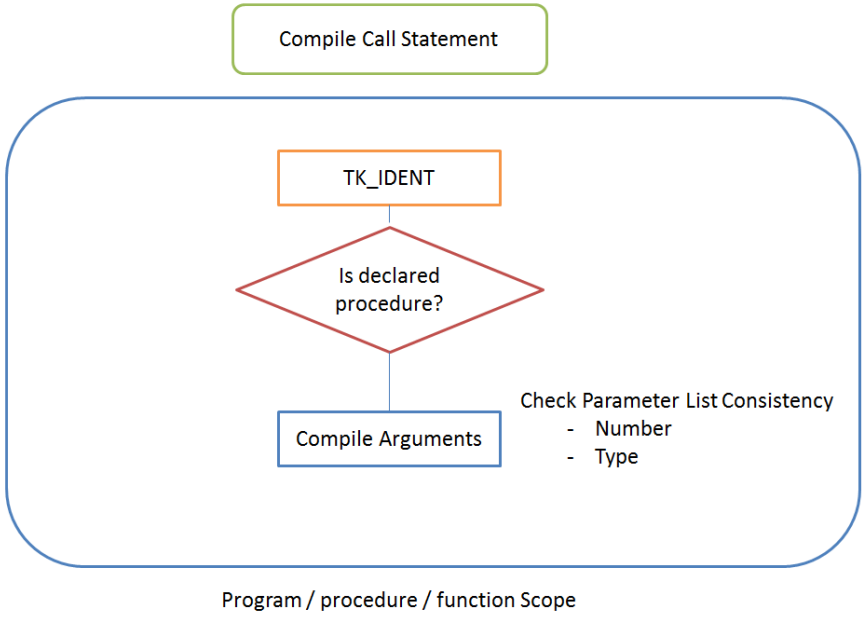
KPL Compiler Scheme - Nguyen Truong Minh - Hanoi University of Science and Technology 10

2. Điều kiện trong lệnh if, vòng lặp for, while

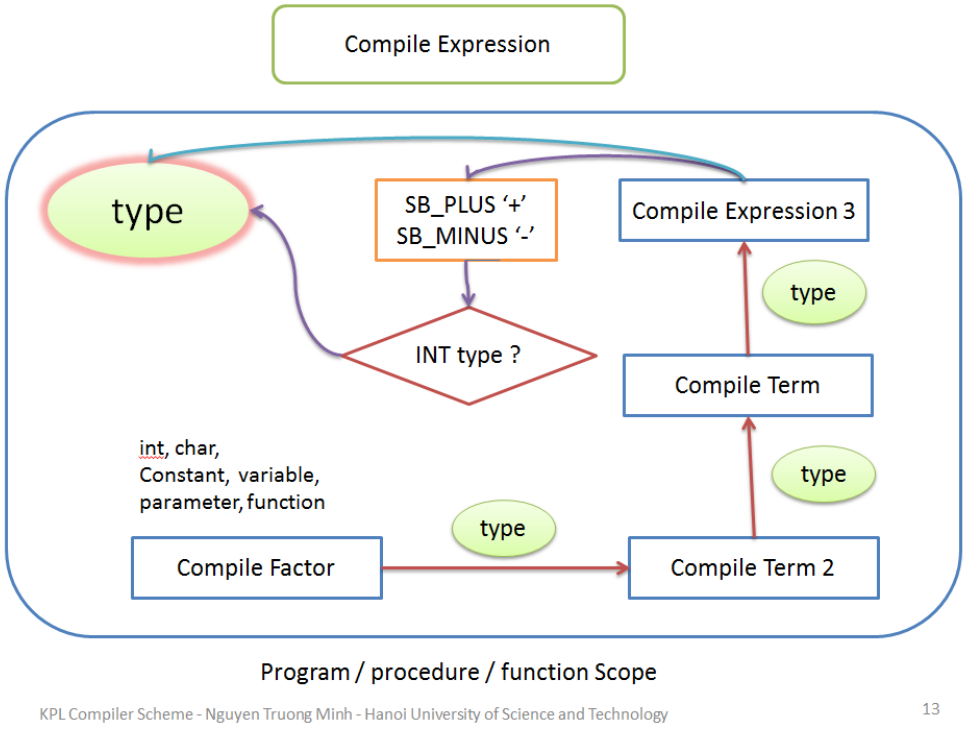


KPL Compiler Scheme - Nguyen Truong Minh - Hanoi University of Science and Technology 12

3. Truyền tham số cho hàm.



4. Các phép toán trong khi compile expression.



III - Sinh mã

CodeGen.h: các tham số của bản ghi hoạt động.
CodeGen.c
1. ComputeNestedLevel.
2. Lấy các thông tin(địa chỉ, giá trị) của biến: khi tìm thông tin về biến cần tính đến phạm vi của nó.

Các biến cục bộ được lấy ở frame hiện tại.

Các biến phi cục bộ được lấy theo các StaticLink với cấp độ lấy theo "độ sâu" của phạm vi hiện tại so với phạm vi của biến. Sử dụng hàm: ComputerNestedLevel(Scope*)

3. Lấy thông tin(địa chỉ, giá trị) của tham số:khi tìm thông tin về tham số cần tính đến phạm vi của nó. Sự khác nhau giữa biến và tham số là ở chỗ việc lấy đía chỉ/giá trị của tham số còn phụ thuộc vào việc tham số đó là tham trị (truyền giá trị) hay tham biến (truyền qua địa chỉ).

2.1. Tham trị: lấy địa chỉ/ giá trị giống như biến.

2.2. Tham biến: địa chỉ của biến chính là giá trị truyền vào cho hàm/thủ tục.

4. Lấy địa chỉ của giá trị trả về:

Giá trị trả về luôn nằm ở offset 0 trên frame, do đó chỉ cần tính thêm số bước chuyển phạm vi ("độ sâu")giống như với biến hay tham số hình thức.

5. Sinh lời gọi hàm/thủ tục

Lời gọi hàm\thủ tục được thực hiện tại lệnh CALL. Hàm còn được gọi trong factor. Trước khi sinh mã cho lệnh CALL cần nạp giá trị cho các tham số hình thức bằng cách:

5.1. Tăng giá trị thanh ghi T lên 4 (để phần ô nhớ cho RV, DL, RA, SL). Đó là các ô nhớ 0, 1, 2, và 3

5.2. Sinh mã cho k tham số thực tế từ địa chỉ. Đó là các ô nhớ 4, 5, ..., 4 + k -1

5.3. Giảm giá trị thanh ghi T đi 4+k.

"Độ sâu" trong lệnh CALL được tính bằng độ sâu tính từ phạm vi định nghĩa hàm\thủ tục được gọi đến phạm vi chứa hàm\thủ tục gọi nó.

5.4. Sau lời gọi hàm\thủ tục, program counter chuyển đến địa chỉ bắt đầu chương trình con

5.5. Lệnh đầu tiên thông thương là lệnh nhảy, để phần ô nhớ cho các hàm\thủ tục cục bộ.

5.6. Lệnh tiếp theo là lệnh nhảy để phần ô nhớ cho frame của hàm\thủ tục hiện tại.

5.7. Khi kết thúc hàm\thủ tục, toàn bộ frame được giải phóng bằng cách đặt con trỏ T lên đầu frame.

Đối với hàm, chỉ có ô nhớ chứa giá trị trả về tại offset 0 được giữ lại. Điều này giải thích tại sao ta đặt giá trị trả về tại offset 0

(1) Để đơn giản hóa việc giải phóng bộ nhớ sau khi thoát hàm.

(2) Khi cần truy xuất giá trị trả về ta chỉ cần tìm đến địa chỉ của hàm.

6. Sinh địa chỉ của phần tử của mảng:

Biểu thức truy xuất giá trị của mảng được viết một các logic theo chỉ số. Khi biên dịch, địa chỉ này được chỉ ra cụ thể (chỉ ra địa chỉ không có nghĩa là cấp phát ô nhớ tại địa chỉ ấy). Các hàm cấp phát bộ nhớ động trong C sử dụng cách tính địa chỉ của chính trình biên dịch C.

Địa chỉ này được tính trong hàm compileIndexes(arrayType). Trong arrayType có chứa sẵn số chiều và kiểu của mỗi phần tử.

7. Lệnh nhảy:

Lệnh nhảy này cho phép ta tìm đến vị trí cần đến trong hai trường hợp:

7.1. Nhảy có điều kiện/có điều kiện trong các lệnh if, for, while

7.2. Nhảy không điều kiện tới vị trí khai báo hàm. Lúc khai báo hàm, lệnh nhảy được tạo ra nhằm giữ chỗ cho hàm. Tuy nhiên label cần nhảy tới chưa được xác định ngay mà đợi

đến khi duyệt xong khai báo hàm, thủ tục, biến, hằng, … cục bộ rồi dùng hàm updateJump sửa label. Label mới này chính là vị trí hiện tại của pc trên code buffer. Đây cũng là vị trí bắt đầu của thân chương trình\hàm\thủ tục.

8. Break Point: hỗ trợ gỡ rối chương trình.

- Sinh mã trong bộ Parser

1. Khi compile Program: gọi compile block, sinh lệnh halt khi kết thúc.
2. Compile Block:

(Bước 1) Vị trí hiện tại chính là địa chỉ của chương trình\hàm\thủ tục. Một lệnh nhảy được tạo ra ở đây.
(Bước 2) Duyệt khai báo hàm, thủ tục, biến, hằng, … cục bộ
(Bước 3) Update label của block chính, label mới này chính là vị trí hiện tại của pc trên code buffer.
(Bước 4) Để dành ô nhớ cho bản ghi hoạt động của chương trình\hàm\thủ tục.
(Bước 5) Compile Statements.

3. Compile Function: Sau khi compile block trong function, sinh lệnh exit function. Lệnh này cho phép nhảy tới địa chỉ ghi trong RA của bản ghi hoạt động.
4. Compile Statements:

(Nhánh 1) Lệnh gán:

Bước 1:
Xét loại object của Vế trái: xác định kiểu, load địa chỉ

- Biến: cần sinh lệnh Load Address để lấy địa chỉ của biến / địa chỉ bắt đầu của mảng. Đối với mảng, compile expression để lấy chỉ số của phần tử, load kích thước của mảng, nhân lên rồi cộng với địa chỉ cơ sở ở trên.
- Tham số: lấy địa chỉ tham số.
- Hàm: lấy địa chỉ bản ghi hoạt động (tại offset 0 là RV).

Xét loại object của Vế phải: xác định kiểu, load địa chỉ.

- Thực hiện giống như cách tính biểu thức dựa trên ký pháp Balan dạng hậu tố.

Bước 2: kiểm tra tương thích kiểu giữa hai vế
Bước 3: sinh lệnh gán.
Các hàm LValue và Expression có hiệu ứng phụ là tạo ra ô nhớ mới trên frame chứa địa chỉ giá trị trả về.

(Nhánh 2) Lệnh gọi thủ tục.

Nếu là thủ tục của thư viện (predefined): xác định các tham số, sau đó sinh mã.
Nếu là thủ tục do người dùng định nghĩa:
Bước 1: để dành ô nhớ cho bản ghi hoạt động.
Bước 2: đưa các tham số lên frame. Kiểm tra kiểu xem có tương thích không để bắt đầu.
Bước 3: Chuyển pc về vị trí bắt đầu thủ tục (tại offset 0).
Bước 4: sinh lời gọi hàm với tham số là (1) "độ sâu" của phạm vi hiện tại so với phạm vi khai báo thủ tục, và (2) địa chỉ của thủ tục trong phạm vi mà nó được khai báo.

(Nhánh 3) Lệnh if :

Bước 1: kiểm tra điều kiện. Sinh lệnh nhảy cho biểu thức sau KW_ELSE.
Bước 2: compile statement() . Xong biểu thức sau KW_THEN, sinh lệnh nhảy để thoát lệnh if.
Bước 3: update lệnh nhảy cho biểu thức sau KW_ELSE, label là vị trí hiện tại.
Bước 4: compile Statement(). Xong biểu thức sau KW_ELSE.

Bước 5: sửa lệnh nhảy cho biểu thức sau KW_THEN, label là vị trí hiện tại. Xong lệnh IF.

(Nhánh 4) Lệnh while

Làm hoàn toàn tương tự biểu thức if. Chú ý hai điểm khác nhau:

1. Biểu thức sau KW_ELSE bằng rỗng.
2. Lệnh nhảy để thoát if trở thành lệnh nhảy về vị trí bắt đầu lệnh while.

(Nhánh 5) Lệnh for

Bước 1: Tính địa chỉ chứa giá trị của vế trái biểu thức gán đầu lệnh for.
Bước 2: Sao chép địa chỉ này (chứa biến chạy).
Bước 3: Tính địa chỉ chứa giá trị của vế phải biểu thức gán đầu lệnh for.
Bước 4: Kiểm tra sự hợp kiểu trong biểu thức gán.
Bước 5: Lưu giá trị của vế phải và địa chỉ chứa giá trị của vế trái.
Bước 6: Sao chép địa chỉ chứa giá trị này.
Bước 7: Load giá trị vào frame.
Bước 8: Load giá trị của biểu thứ sau KW_TO.
Bước 9: Kiểm tra sự hợp kiểu với giá trị sau biểu thức KW_TO.
Bước 10: Sinh lệnh so sánh giá trị.
Bước 11: Sinh lệnh nhảy điều kiện sai.
Bước 12: Duyệt thân của lệnh for.
Bước 13: Sao chép địa chỉ chứa giá trị biến chạy hai lần.
Bước 14: Lấy giá trị của biến chạy, load bước nhảy của biến chạy giữa hay vòng lặp.
Bước 15: Cộng giá trị của biến chạy và bước nhảy.
Bước 16: Lưu giá trị vào địa chỉ chứa biến chạy.
Bước 17: Lấy giá trị của biểu thứ sau KW_TO
Bước 18: Nhảy về đầu lệnh for.
Bước 19: Update lệnh nhảy điều kiện sai. Xong vòng lặp for.
Bước 20: tăng biến đến lên 1.

- Ví dụ: Chương trình sau có bao gồm khá đầy đủ các đặc trưng cấu trúc chương trình mà trình dịch KPL cần xử lý, như các hàm lồng nhau, lời gọi hàm, lệnh if, lệnh lặp for.

```
Program Example2; (* Factorial *)

Var n : Integer;

Function F(n : Integer) : Integer;
  Begin
    If n = 0 Then F := 1 Else F := N * F (N - 1);
  End;

Begin
  For n := 1 To 7 Do
    Begin
      Call WriteLn;
      Call WriteI( F(n));
    End;
End. (* Factorial *)
```

```
t = 0;
Code Block từ địa chỉ 0 đến địa chỉ 48.
0:  J 22
1:  J 2
2:  INT 5   // Để dành ô nhớ cho activation record
3:  LV 0,4  // N
4:  LC 0    // 0
5:  EQ            // Nếu N = 0

6:  FJ 11   // Nhay điều kiện sai
7:  LA 0,0  // Biểu thức sau KW_THEN. Load địa chỉ chứa giá trị trả về.
9:  ST            // lưu giá trị.
10:  J 21         // Exit function. J to 21.
11:  LA 0,0 // Biểu thức sau KW_ELSE. Load địa chỉ chứa giá trị trả về.
12:  LV 0,4       // Tham sô N của hàm hiện tại

13:  INT 4        // Để dành vùng nhớ cho actiation record
14:  LV 0,4       // Tham số N của hàm mới
15:  LC 1
16:  SB           // N - 1
17:  DCT 5        // Hết tham số. Quay trở lại bản ghi hoạt động.
18:  CALL 1,1     // Để dành ô nhớ cho activation record
// s[t+2] := b; s[t+3] := pc; s[t+4]:= base(p); b:=t+1;pc:=q;
// (DL) s[8] := base của caller = 2, giúp ta khôi phục ngữ cảnh của caller.
// (RA) s[9] := 18 ; giúp ta biết tiếp tục chương trình từ đâu.
// (SL) s[10] = base(1) địa chỉ của phạm vi chứa khai báo hàm trên code
block. (nested level = 1)
//  base mới(địa chỉ tuyệt đối của activation record của hàm, trên frame)= 7;
// Giúp cho việc tính LV dựa trên địa chỉ tương đối. Đồng thời là RV.
// pc = 1 (trên bộ đệm code)
// Khi thoát hàm, giá trị trả về nằm ở ô nhớ hiện tại.
19:  ML                 // (N-1) * giá trị trả về của hàm mới
20:  ST                 // Lưu vào giá trị trả về của hàm hiện tại.
21:  EF                 // Thoát hàm.
22:  INT 5        // Để dành ô nhớ cho activation record
23:  LA 0,4       // Biến cục bộ N của chương trình.
24:  CV
25:  LC 1         //
26:  ST                 // N = 1
27:  CV
28:  LI                 // lôi N ra. Ta đang ở trong vòng lặp rồi.
// Không dùng biểu thức trước được mà phải load riêng.
29:  LC 7         //
30:  LE                 // N < 7
31:  FJ 47        // Thoát vòng lặp for
32:  WLN          // Chưa nhảy, chưa thoát vòng lặp. New line
33:  INT 4        // Để dành ô nhớ cho activation record
34:  LV 0,4       // lấy N thuộc phạm vi chương trình.
35:  DCT 5
36:  CALL 0,1     // gọi hàm. Nested level = 0 cho biết hàm được khai báo ở
                  phạm vi hiện tại.
                  // Địa chỉ bắt đầu hàm trong phạm vi này là 1.
37:  WRI          // viết giá trị trả về của hàm.
                  // Hàm thư viện, không phải tạo bản ghi hoạt động, chỉ cần
load tham số.
```

```
38:  CV
39:  CV
40:  LI
41:  LC 1
42:  AD          // N : = N + 1
43:  ST
44:  CV
45:  LI
46:  J 29        // Vòng lặp mới
47:  DCT 1       //
48:  HL          // thoát chương trình
```