ПРИЛОЖЕНИЕ А

Программный проект

А.1. Введение

В этом приложении вашему вниманию предлагается программное упражнение для практических занятий, соответствующих учебному курсу по разработке компиляторов, основанному на данной книге. Упражнение состоит в реализации базовых компонентов компилятора для подмножества языка программирования Pascal. Это подмножество минимально, но обеспечивает возможность создания таких программ, как использованная в разделе 7.1 процедура рекурсивной сортировки. То, что данный язык является подмножеством существующего, дает определенные преимущества. Смысл программы, написанной с использованием данного подмножества, определяется семантикой Pascal [214]. При доступности компилятора Pascal он может использоваться для проверки работы компилятора, создаваемого в качестве упражнения. Конструкции данного подмножества встречаются во многих языках программирования, так что соответствующее упражнение может быть легко переформулировано для использования другого языка программирования в случае недоступности компилятора Pascal.

А.2. Структура программы

Программа состоит из последовательности объявлений глобальных данных, последовательности процедур и объявлений функций, а также составной инструкции, представляющей собой "основную программу". Глобальные данные хранятся в статически выделяемой памяти. Данные, локальные по отношению к процедурам и функциям, размещаются в стеке. Рекурсия разрешена, и параметры передаются по ссылке. Процедуры `read` и `write` обеспечивает компилятор.

На рис. А.1 приведен пример программы. Имя программы — `example`, а `input` и `output` — имена файлов, используемые процедурами соответственно `read` и `write`.

```
program example(input, output);
var x, y: integer;
function gcd(a, b: integer): integer;
begin
    if b = 0 then gcd := a;
    else gcd := gcd(b, a mod b)
end;

begin
    read(x, y);
    write(gcd(x, y))
end.
```

Рис. А.1. Пример программы

A.3. Синтаксис подмножества Pascal

Ниже приведена LALR(1)-грамматика подмножества Pascal. Данная грамматика может быть модифицирована для синтаксического анализа методом рекурсивного спуска путем устранения левой рекурсии, как описано в разделах 2.4 и 4.3. Синтаксический анализатор на основе приоритета операторов для выражений может быть построен замещением **relop**, **addop** и **mulop** и устранением ε-продукций.

Добавление продукции

statement → **if** *expression* **then** *statement*

приводит к неоднозначности "кочующего **else**", которую можно устранить описанным в разделе 4.3 способом (см. также пример 4.19 в случае применения предиктивного синтаксического анализатора).

Синтаксического отличия между простой переменной и вызовом функции без параметров нет. Оба генерируются продукцией

factor → **id**

Таким образом, присвоение `a := b` устанавливает `a` равным значению, возвращаемому функцией `b`, если `b` представляет собой функцию.

program →
  program id (*identifier_list*) **;**
  declarations
  subprogram_declarations
  compound_statement
  .

identifier_list →
  id
  | *identifier_list* **, id**

declarations →
  declarations **var** *identifier_list* **:** *type* **;**
  | ε

type →
  standard_type
  | **array [num .. num] of** *standard_type*

standard_type →
  integer
  | **real**

subprogram_declarations →
  subprogram_declarations subprogram_declaration **;**
  | ε

subprogram_declaration →
  subprogram_head declarations compound_statement

subprogram_head →
  function id *arguments* **:** *standard_type* **;**
  | **procedure id** *arguments* **;**

arguments →
    (*parameter_list*)
    | ε

parameter_list →
    identifier_list : *type*
    | *parameter_list* ; *identifier_list* : *type*

compound_statement →
    begin
    optional_statements
    end

optional_statements →
    statement_list
    | ε

statement_list →
    statement
    | *statement_list* ; *statement*

statement →
    variable **assignop** *expression*
    | *procedure_statement*
    | *compound_statement*
    | **if** *expression* **then** *statement* **else** *statement*
    | **while** *expression* **do** *statement*

variable →
    id
    | **id** [*expression*]

procedure_statement →
    id
    | **id** (*expression_list*)

expression_list →
    expression
    | *expression_list* , *expression*

expression →
    simple_expression
    | *simple_expression* **relop** *simple_expression*

simple_expression →
    term
    | *sign term*
    | *simple_expression* **addop** *term*

term →
    factor
    | *term* **mulop** *factor*

factor →
    id
    | **id** (*expression_list*)

| **num**
| (*expression*)
| **not** *factor*

sign →
+ | -

A.4. Лексические соглашения

Система записи токенов взята из раздела 3.3.

1. Комментарии заключаются в фигурные скобки { и }. Они не могут содержать {. Комментарии могут располагаться после любого токена.
2. Пробелы между токенами необязательны, однако ключевые слова должны быть окружены пробелами, символами новой строки, находиться в начале программы или завершаться заключительной точкой.
3. Токен **id** идентификатора представляет собой букву, за которой следуют буквы или цифры.

letter → `[a-zA-Z]`
digit → `[0-9]`
id → **letter (letter | digit) ***

При реализации может быть ограничена допустимая длина идентификатора.

4. Токен **num** соответствует беззнаковым числам (см. пример 3.5).

digits → **digit digit***
optional_fraction → **. digits | ε**
optional_exponent → **(E (+ | - | ε) digits) | ε**
num → **digits optional_fraction optional_exponent**

5. Ключевые слова зарезервированы и в грамматике обозначаются полужирным шрифтом.
6. Операторы отношения (**relop**) представляют собой =, <>, <, <=, >= и >. Обратите внимание, что <> означает ≠.
7. Операторами **addop** являются +, - и `or`.
8. Операторы **mulop** представляют собой *, /, `div`, `mod` и `and`.
9. Лексемой токена **assignop** является :=.

A.5. Предлагаемые упражнения

Программный практикум для односеместрового курса состоит в написании интерпретатора для определенного выше языка или для аналогичного подмножества другого языка высокого уровня. Проект включает трансляцию исходной программы в промежуточное представление (в виде четверок или кода стековой машины) и интерпретацию промежуточного представления. Мы предлагаем следующий порядок создания модулей, отличающийся от порядка их работы при компиляции, поскольку удобно иметь работающий интерпретатор для отладки других компонентов компилятора.

1. *Разработка механизма таблицы символов.* Решение об организации таблицы символов должно обеспечивать сбор информации об именах, оставляя структуру записи таблицы символов по возможности более гибкой. Напишите подпрограммы для следующих действий.

   i) Поиск имени в таблице символов, создание новой записи для отсутствующего имени и возврата указателя на запись для имени в обоих случаях.

   ii) Удаление из таблицы символов всех имен, локальных для данной процедуры.

2. *Создание интерпретатора для четверок.* Вопрос о точном множестве четверок в настоящий момент может оставаться открытым, но оно должно включать арифметические операторы и инструкции условных переходов, соответствующие множеству операторов языка. Кроме того, множество включает логические операторы, если условия вычисляются арифметически, а не по положениям в программе. Кроме того, необходимы "четверки" для преобразования целых чисел в действительные, для маркировки начала и конца процедур, а также для передачи параметров и вызова процедур.

   Необходимо также разработать вызывающую последовательность и организацию среды времени выполнения интерпретируемой программы. Простая стековая организация, рассмотренная в разделе 7.3, вполне подходит для нашего языка, поскольку в нем не разрешены вложенные объявления процедур, так что все переменные либо являются глобальными, объявленными на уровне всей программы, либо локальны по отношению к простой процедуре.

   Для простоты вместо интерпретатора может быть использован другой язык высокого уровня. Каждая четверка может являться инструкцией языка высокого уровня типа C или даже Pascal. Вывод компилятора в таком случае представляет собой последовательность инструкций C, которая может быть скомпилирована существующим компилятором C.

3. *Создание лексического анализатора.* Выберите внутренние коды для токенов. Решите, как константы будут представляться в компиляторе. Решите задачу подсчета считанных строк для корректного отображения места обнаружения ошибки. Учтите, что при необходимости лексический анализатор должен выводить листинг исходной программы. Разработайте программу для внесения ключевых слов в таблицу символов. Разработайте лексический анализатор как подпрограмму, вызываемую синтаксическим анализатором, и возвращающую пару (токен, значение атрибута). На этом этапе при обнаружении лексическим анализатором ошибки ее обработка может сводиться к выводу соответствующего сообщения и прекращению работы программы.

4. *Разработка семантических действий.* Напишите семантические программы для генерации четверок. Грамматика потребует внесения изменений для упрощения трансляции — обратитесь к разделам 5.5 и 5.6 за примерами таких изменений грамматики. Обеспечьте на этом этапе выполнение семантического анализа, преобразуя при необходимости целые числа в действительные.

5. *Разработка синтаксического анализатора.* Если у вас имеется генератор LALR-анализаторов, это значительно упростит вашу задачу. Если такой генератор способен обрабатывать неоднозначности грамматики, как Yacc, то нетерминалы, означающие выражения, могут быть объединены. Кроме того, неоднозначность "кочующего **else**" может быть разрешена путем переноса при возникновении конфликта переноса/свертки.

6. *Создание подпрограмм обработки ошибок.* Разработайте восстановление после лексических и синтаксических ошибок. Обеспечьте вывод диагностики для лексических, синтаксических и семантических ошибок.

7. *Вычисления*. Программа на рис. А.1 может служить в качестве простой тестовой программы. Другой тестовой программой может быть программа на рис. 7.1. Код для функции `partition` на этом рисунке соответствует помеченному фрагменту на рис. 10.2. Если у вас есть такая возможность, пропустите ваш компилятор через профайлер. Определите подпрограммы, занимающие основное время работы. Какие модули следует модифицировать для ускорения работы вашего компилятора?

А.6. Эволюция интерпретатора

Альтернативный подход к построению интерпретатора для нашего языка состоит в реализации настольного калькулятора, т.е. интерпретатора для выражений. Постепенное добавление конструкций к языкам этого калькулятора приведет к получению интерпретатора для всего языка. Аналогичный подход используется в книге [247]. Предлагается следующий порядок добавления конструкций.

1. *Трансляция выражений в постфиксную запись*. Используя метод рекурсивного спуска, описанный в главе 2, "Простой однопроходный компилятор", либо генератор синтаксических анализаторов, освойтесь со средой программирования и напишите транслятор простых арифметических выражений в постфиксную запись.
2. *Добавление лексического анализатора*. Обеспечьте работу транслятора из предыдущего пункта с ключевыми словами, идентификаторами и числами. Переделайте транслятор для генерации кода стековой машины или четверок.
3. *Создание интерпретатора для промежуточного представления*. Как говорилось в разделе А.5, вместо интерпретатора может использоваться язык высокого уровня. В настоящий момент интерпретатор должен поддерживать только арифметические операции, присвоение и операции ввода-вывода. Расширьте язык путем добавления объявлений глобальных переменных, присвоений и вызовов процедур `read` и `write`. Наличие этих конструкций обеспечивает возможность тестирования интерпретатора.
4. *Добавление инструкций*. Программа на нашем языке теперь состоит из основной программы без объявлений подпрограмм. Протестируйте транслятор и интерпретатор.
5. *Добавление процедур и функций*. Таблица символов должна теперь обеспечивать ограничение области видимости идентификаторов телом процедуры. Разработайте последовательность вызова (простая организация стека из раздела 7.3 вполне адекватна поставленной задаче). Расширьте интерпретатор для поддержания последовательности вызова.

А.7. Расширения

Имеется ряд возможностей, которые могут быть добавлены в язык без существенного повышения сложности компиляции.

1. Многомерные массивы.
2. Инструкции `for` и `case`.
3. Блочная структура.
4. Структуры записей.

Если позволяет время — добавьте одно или несколько из этих расширений в ваш компилятор.